, DOTTY, AND THE AWESOME POWER OF 'inline'

Alexander Ioffe

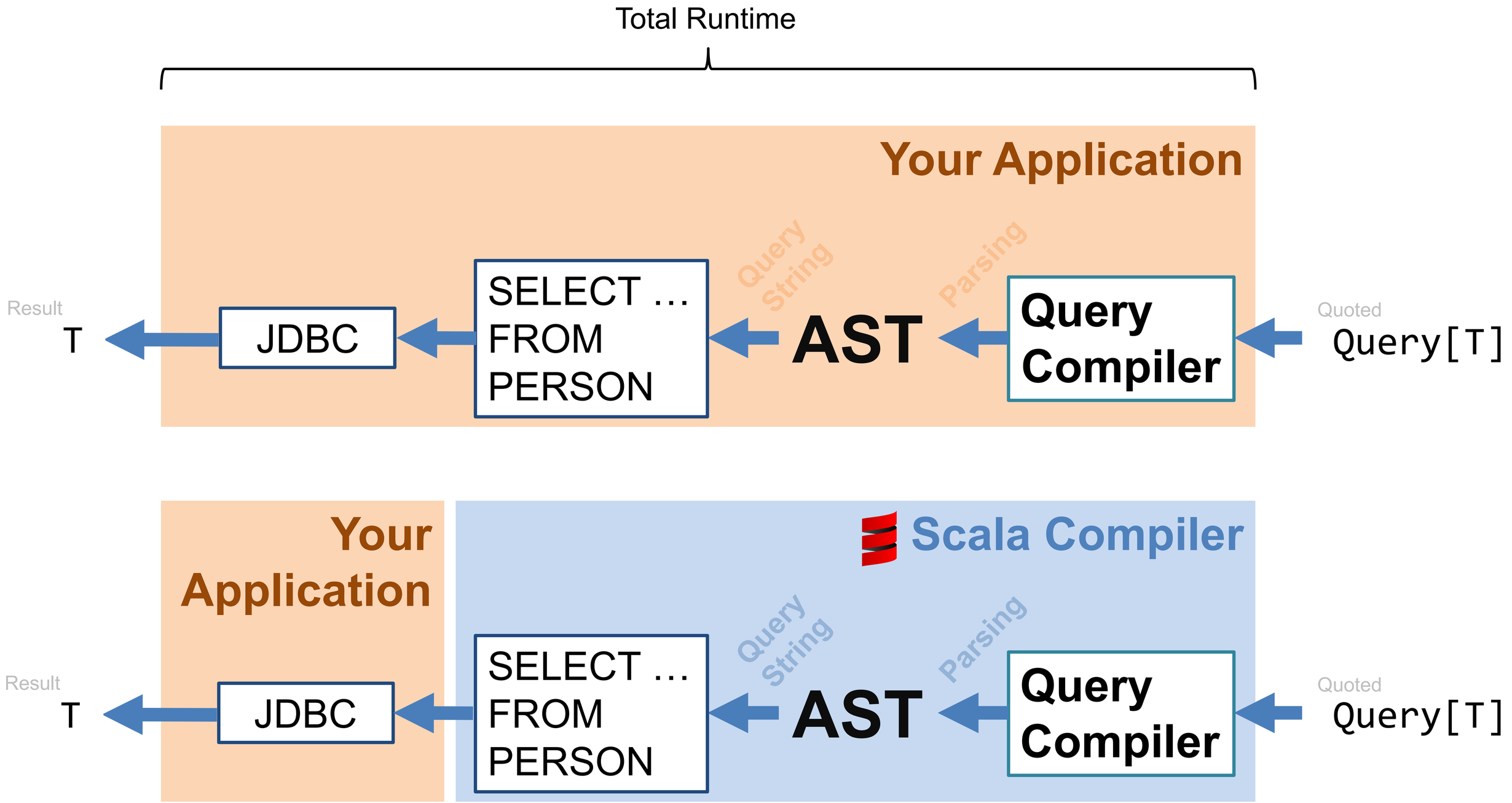
@deusaquilus

https://github.com/deusaquilus/dotty_test

What is Quill?

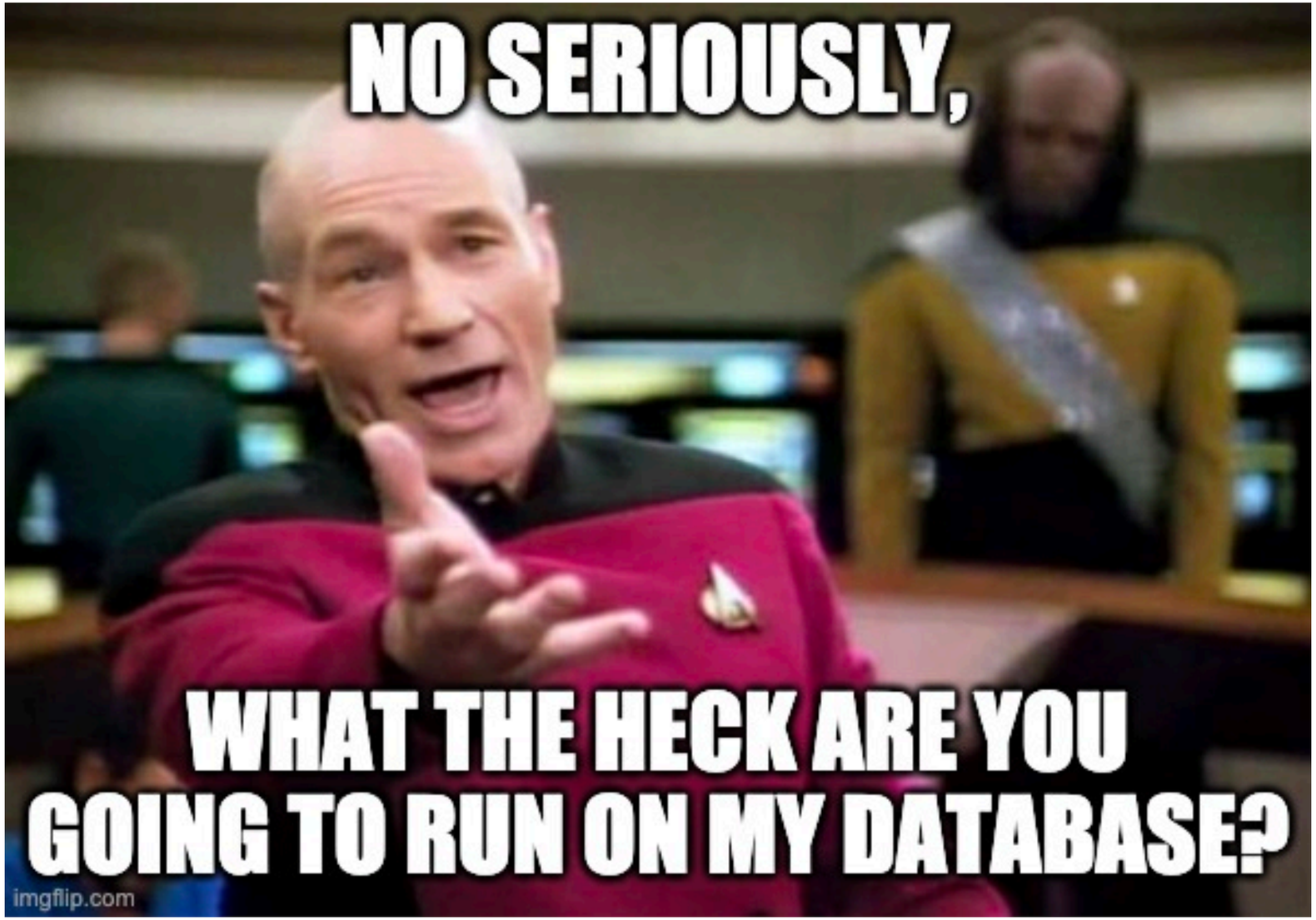
```
case class Person(name: String, age: Int)

val q = query {
  query[Person].filter(p => p.name == "Joe")
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT p.name, p.age FROM Person p WHERE p.name = 'Joe'
```


Total Runtime
Your Application
Result
T
JDBC
SELECT …
FROM
PERSON
Query String
AST
Parsing
Query Compiler
Quoted
Query[T]
Your Application
Scala Compiler
Result
T
JDBC
SELECT …
FROM
PERSON
Query String
AST
Parsing
Query Compiler
Quoted
Query[T]

Then Your DBA Asks…

In a fit of Desperation at 3:00 AM



Bring on the Complexity…

```
val q = quote {
  for {
    o1 <- query[Country]
    c1 <- query[Company].join(c1 => c1.countryId == o1.id)
    p1 <- query[Person].join(p1 => p1.companyId == c1.id)
    af <- query[ClientAffiliation].leftJoin(af => af.of == p1.id)
    p2 <- query[Person].leftJoin(p2 => af.map(_.host).exists(v => v == p2.id))
  } yield (o1, c1, p2, af, p2)
}
run(q)
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT o1.id, o1.name, o1.population, c1.id, c1.name, c1.countryId,
p1.id, p1.name, p1.age, p1.companyId, af.host, af.of,  p2.id, p2.name,
p2.age, p2.companyId FROM Country o1
INNER JOIN Company c1 ON c1.countryId = o1.id
INNER JOIN Person p1 ON p1.companyId = c1.id
LEFT JOIN ClientAffiliation af ON af.of = p1.id
LEFT JOIN Person p2 ON af.host = p2.id
```

Limitation - Must Lift Runtime Values

```scala
case class Person(name: String, age: Int)

val q = query {
  query[Person].filter(p => p.name == lift(runtimeVar))
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
SELECT p.name, p.age FROM Person p WHERE p.name = ?
```

Limitation - ~~Runtime Switches~~

```scala
case class Person(name: String, age: Int)

val q =
    if (someRuntimeCondition)
      quote { query[Person].filter(p => p.name == "Joe") }
    else
      quote { query[Person] }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

Limitation - ~~Dynamic Filters~~

```scala
case class Person(name: String, age: Int)

val filters = Map("name" -> "Joe", "age" -> "22")
val q = query {
  query[Person].filter(p =>
    col(p, "name") == lift(filters("name")) AND
    col(p, "age") == lift(filters("age"))
  )
}
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

Limitation - ~~Encapsulating Methods~~

```
case class Person(name: String, age: Int)

val q = quote {
      joes(query[Person])
    }

def joes(q: Query[People]) =
  q.filter(p => p.name == "Joe")
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

This...

```
case class Person(name: String, age: Int)

val q: Quoted[Query[Person]] =
      quote { query[Person].filter(p => p.name == "Joe") }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```

… Is Really This

```
case class Person(name: String, age: Int)

val q: AnyRef with Quoted[Query[Person]] { def quoted… def lifted… } =
      quote { query[Person].filter(p => p.name == "Joe") }
```

```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```


```
sbt:my-project> compile
[info] Compiling 1 Scala source to /Users...
Dynamic Query
```


inline

Before We Get Started...

```scala
val greeting = "Hello"
PrintMac(greeting)
```

```
[info] Compiling 1 Scala source to /Users...
Inlined(
  Thicket(List()),
  List(),
  Ident(greeting)
)
```

```scala
inline def greeting = "Hello"
PrintMac(greeting)
```

```
[info] Compiling 1 Scala source to /Users...
Inlined(
  Thicket(List()),
  List(),
  Inlined(
    Ident(greeting),
    List(),
    Literal(("Hello")),
  )
)
```

```scala
inline def greeting = "Hello"
inline def suffix = " world"
inline def combo = greeting + suffix
PrintMac(combo)
```

```
Inlined(                                                    Inlined(<originalArg>, <binds>, <expansion>)
  Thicket(List()), List(),
  Inlined(
    Ident(combo), List(),
    Typed(
      Apply(
        Select(
          Inlined(Ident(greeting), List(), Typed(Literal(("Hello")),TypeStuff())),
          +
        ),
        List(Inlined(Ident(suffix),List(),Typed(Literal((" world")),TypeStuff())))
      ),
      TypeStuff()
    )
  )
)
```

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
inline def passThrough(inline str: String): String = ${ passThroughImpl('str) }
def passThroughImpl(str: Expr[String])(using qctx: QuoteContext): Expr[String] = {
  import qctx.tasty._
  println(pprint(str.unseal))
  str
}
```

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello + " world”)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

```
Inlined(Thicket(List()),List(),Apply(Select(
      Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world),List(),
        Inlined(Apply(Select(Ident(Mac), passThrough),List(Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello")))),List(),
                    Inlined(Thicket(List()), List(), Literal(("hello")))),
                )
              ), + ), List(Literal(("world")))))
            ),
            List(),
            Inlined(Thicket(List()),List(),Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello")))),List(),
                  Inlined(Thicket(List()), List(), Literal(("hello")))),
                ), + ), List(Literal(("world")))))),
        ),
      )), + ), List(Literal((" today")))
  )
)
```

Inlined(<originalArg>, <binds>, <expansion>)

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello + " world”)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

```
Inlined(Thicket(List()),List(),Apply(Select(
      Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world),List(),
        Inlined(Apply(Select(Ident(Mac), passThrough),List(Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))),List(),
                    Inlined(Thicket(List()), List(), Literal(("hello")))),
                )
              ), + ), List(Literal(("world")))))
            ),
            List(),
            Inlined(Thicket(List()),List(),Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello"))))),List(),
                  Inlined(Thicket(List()), List(), Literal(("hello")))),
                ), + ), List(Literal(("world"))))),
          ),
        )), + ), List(Literal((" today")))
  )
)
```

Applying In

Inlined(<originalArg>, <binds>, <expansion>)

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello + " world”)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

```
Inlined(Thicket(List()),List(),Apply(Select(
      Inlined(Select(Apply(Select(New(Ident(Space)), <init>), List()), world),List(),
        Inlined(Apply(Select(Ident(Mac), passThrough),List(Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello")))),List(),
                    Inlined(Thicket(List()), List(), Literal(("hello")))),
                )
              ), + ), List(Literal(("world")))))
            ),
            List(),
            Inlined(Thicket(List()),List(),Apply(Select(
              Inlined(Select(Apply(Select(New(Ident(InnerSpace)), <init>), List()), hello),List(),
                Inlined(Apply(Select(Ident(Mac), passThrough), List(Literal(("hello")))),List(),
                  Inlined(Thicket(List()), List(), Literal(("hello")))),
                ), + ), List(Literal(("world"))))),
          ),
        )), + ), List(Literal((" today")))
  )
)
```

Applying In

Expanding Out

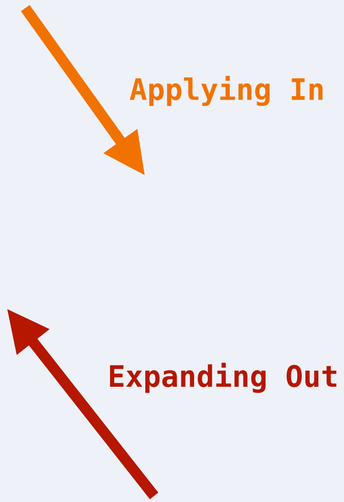
Inlined(<originalArg>, <binds>, <expansion>)

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello + " world”)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

Built-in Lineage



Inlined(<originalArg>, <binds>, <expansion>)

Phase Consistency Principle

A.K.A Passing the Buck... in a good way!

```
class Space {
  class InnerSpace {
    inline def hello = Mac.passThrough("hello")
  }
  inline def world = Mac.passThrough(new InnerSpace().hello + " world”)
}
inline def today = Mac.passThrough(new Space().world + " today")
println(today)
```

Hygienic ⊃ Built-in Lineage

In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def p =
  query[Person]
```

```
inline def q = quote {
  query[Person]
}
```

```
Inlined(EntityQuery[Person])
```

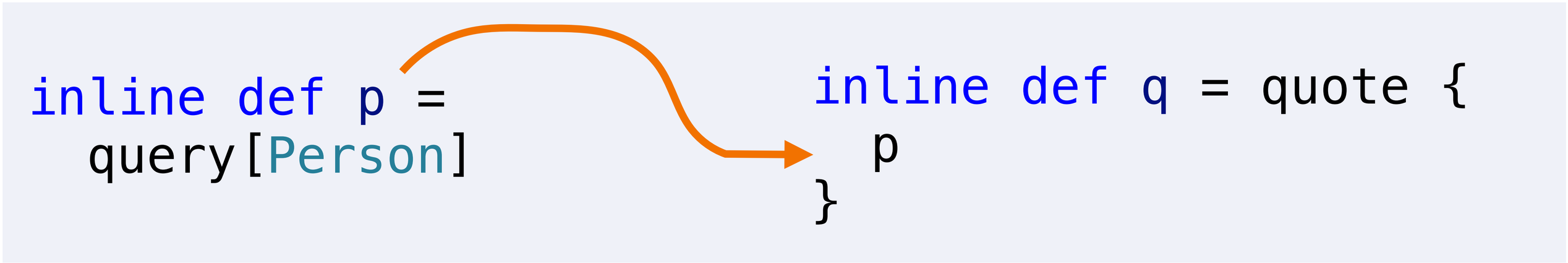
```
Inlined(Quoted(EntityQuery[Person] ⋯)
```

```
inline def p =
  query[Person]
```

```
inline def q = quote {
  query[Person]
}
```

```
inline def p = Inlined(EntityQuery[Person])
```

```
Inlined(p ...)
```



```
inline def p = Inlined(EntityQuery[Person])
```

```
Inlined(Inlined(EntityQuery[Person]) ⋯)
```

THE COMPILER DOES THIS!

```
inline def p =
  query[Person]
```

```
inline def q = quote {
  p
}
```

```
inline def p = Inlin
```

```
ityQuery[Person]) ⋯)
```

KEEP CALM

THROW
WHITEBOX
MACROS
OUT OF
A WINDOW

```
inline def p =
  query[Person]
```

```
ef q = quote {
```

```
inline c
                                                    on]) ⋯)

inline                                                te {
  quer
```

In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def p =
  query[Person]
```

```
inline def q = quote {
  p
}
```

```
sbt:dotty-simple> compile
[info] Compiling 1 Scala source to /Users...
SELECT x.name, x.age FROM Person x
```

In Scala 3 Quill Quotes will be Inline

... and life will be awesome!

```
inline def onlyJoes =
  (p: Person) => p.name == "Joe"
```

```
inline def q = quote {
  query[Person].filter(onlyJoes)
}
```

```
List(Person("Joe", 22), Person("Jack", 33)).filter(onlyJoes)
```

```
[info] running miniquill.MiniExample3_InlineFilter
List(Person(Joe,22))
```

MiniExample3_InlineFilter.scala

```scala
inline def joes(inline q: Query[Person]) =
  q.filter(p => p.name == "Joe")
```

```scala
inline def q = quote {
  joes(query[Person])
}
```

```
sbt:dotty-simple> compile
[info] Compiling 1 Scala source to /Users...
SELECT x.name, x.age FROM Person x WHERE p.name = 'Joe'
```

```scala
inline def joes(inline q: Query[Person], inline filter: Boolean) =
  inline if (filter)
    q.filter(p => p.name == "Joe")
  else
    q
```

```scala
inline def q = quote {
  joes(query[Person], true)
}
```

```
sbt:dotty-simple> compile
[info] Compiling 1 Scala source to /Users...
SELECT x.name, x.age FROM Person x WHERE p.name = 'Joe'
```

MiniExample1_Joes.scala

```scala
inline def joes(inline q: Query[Person], inline filter: Boolean) =
  inline if (filter)
    q.filter(p => p.name == "Joe")
  else
    q
```

```scala
inline def q = quote {
  joes(query[Person], false)
}
```

```
sbt:dotty-simple> compile
[info] Compiling 1 Scala source to /Users...
SELECT x.name, x.age FROM Person x
```

MiniExample1_Joes.scala

Filters with Http4s

```
get: /app/people?isJoe=true
```

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      val q =
        if (isJoe)
          quote { query[Person].filter(p => p.name == "Joe") }
        else
          quote { query[Person] }
      run(q)
    })).transact(xa).map(_.asJson))
```

```
[info] Compiling 1 Scala source to /Users
Dynamic Query
```

```
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      val q1 = quote { query[Person].filter(p => p.name == "Joe") }
      val q2 = quote { query[Person] }
      if (isJoe) run(q) else run(q)
    })).transact(xa).map(_.asJson))
```

```
[info] Compiling 1 Scala source to /Users
SELECT p.name, p.age FROM Person p
  WHERE p.name = 'Joe'
SELECT p.name, p.age FROM Person p
```

Filters with Http4s

```
get: /app/people?isJoe=true
```

```scala
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? JoesMatcher(isJoe: Boolean) =>
    Ok(stream(joinTables(lift(role), {
      inline def people(inline isJoeInline: Boolean) =
        inline if (isJoeInline)
          query[Person].filter(p => p.name == "Joe")
        else
          query[Person]
      if (isJoe) run(q, true) else run(q, false)
    })).transact(xa).map(_.asJson))
```

```
[info] Compiling 1 Scala source to /Users
SELECT p.name, p.age FROM Person p
  WHERE p.name = 'Joe'
SELECT p.name, p.age FROM Person p
```

```
inline def liftOrAny(inline field: String, inline filter: Option[String]) =
   lift(filter.getOrElse(null)) == field ||
     lift(filter.getOrElse(null)) == null
```

```
val runtimeValue = Some("Joe")

inline def q = quote {
  query[Person].filter(p => liftOrAny(p.name, runtimeValue))
}
```

```
SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL
```

MiniExample2_LiftOrAny.scala

```
inline def liftOrAny(inline field: String, inline filter: Option[String]) =
   lift(filter.getOrElse(null)) == field ||
     lift(filter.getOrElse(null)) == null
```

```
val runtimeValue = Some("Joe")

inline def q = quote {
  query[Person].filter(p => liftOrAny(p.name, runtimeValue))
}
```

```
SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL
```

at runtime...

filter

Some("Joe") → WHERE 'Joe' = p.name OR 'Joe' IS NULL

None → WHERE null = p.name OR null IS NULL

```scala
inline def liftOrAny(inline field: String, inline filter: Option[String]) =
   lift(filter.getOrElse(null)) == field ||
     lift(filter.getOrElse(null)) == null
```

```
get: /app/people or /app/people?name=Joe
```

```scala
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" / NameMatcher(nameOpt: Option[String]) =>
    Ok(stream(joinTables(lift(role), {
      run { query[Person].filter(p => liftOrAny(p.name, nameOpt) }
    })).transact(xa).map(_.asJson))
```

```
SELECT p.name, p.age FROM Person p WHERE ? = p.name OR ? IS NULL
```

at runtime...

filter

Some("Joe") → WHERE 'Joe' = p.name OR 'Joe' IS NULL

None → WHERE null = p.name OR null IS NULL

```
extension [T](inline q: EntityQuery[T]):
  inline def filterByKeys(inline map: Map[String, String]) =
    q.filter(p => MapProc[T, PrepareRow](p, map, null, (a, b) => (a == b) || (b == (null) ) ))
```

```
case class Person(firstName: String, lastName: String, age: Int)


val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")

inline def q = quote {
  query[Person].filterByKeys(values)
}
```

```
SELECT p.firstName, p.lastName, p.age
FROM Person p
WHERE
  (p.firstName = ? OR ? IS NULL) AND
  (p.lastName = ? OR ? IS NULL) AND
  (p.age = ? OR ? IS NULL) AND
  true
```

MiniExample_LiftByKeys.scala

```
inline def apply[Person, PrepareRow](
  entity: Person, map: Map[String, String], default:String, eachField: (String, String) => Boolean): Boolean
```

```scala
case class Person(firstName: String, lastName: String, age: Int)


val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")

inline def q = quote {
  query[Person].filterByKeys(values)
}
```

```
SELECT p.firstName, p.lastName, p.age
FROM Person p
WHERE
  (p.firstName = values.get("firstName") OR values.get("firstName") IS NULL) AND
  (p.lastName = values.get("lastName") OR values.get("lastName") IS NULL) AND
  (p.age = values.get("age") OR values.get("age") IS NULL) AND
  true
```

MiniExample_LiftByKeys.scala

```
inline def apply[Person, PrepareRow](
  entity: Person, map: Map[String, String], default:String, eachField: (String, String) => Boolean): Boolean
```

```scala
case class Person(firstName: String, lastName: String, age: Int)


val values: Map[String, String] = Map("firstName" -> "Joe", "age" -> "22")

inline def q = quote {
  query[Person].filterByKeys(values)
}
```

```
SELECT p.firstName, p.lastName, p.age
FROM Person p
WHERE
  (p.firstName = 'Joe' OR 'Joe' IS NULL) AND
  (p.lastName = null OR null IS NULL) AND
  (p.age = '22' OR '22' IS NULL) AND
  true
```

MiniExample_LiftByKeys.scala

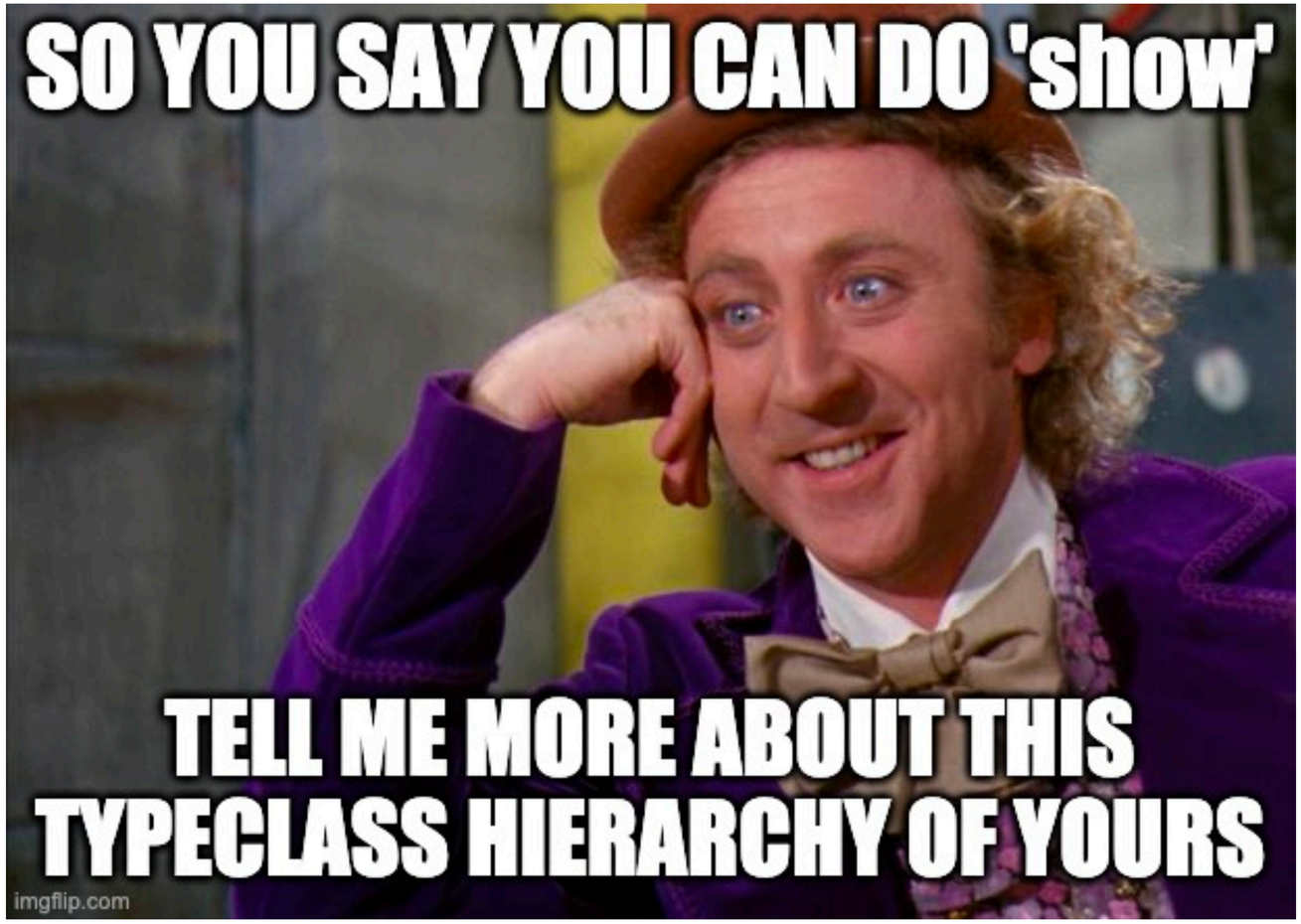
```
get: /app/people?firstName=Joe&age=22
```

```scala
val service = HttpRoutes.of[IO] {
  case GET -> Root / "people" :? (multiFilters: Map[String, Seq[String]]) =>
    Ok(stream(joinTables(lift(role), {
      val filters: Map[String, String] = multiFilters.flatMap((k, v) => (k, v.take(1))
      val q = quote {
        query[Person].filterByKeys(filters)
      }
      run(q)
    })).transact(xa).map(_.asJson))
```

```
SELECT p.firstName, p.lastName, p.age
FROM Person p
WHERE
  (p.firstName = 'Joe' OR 'Joe' IS NULL) AND
  (p.lastName = null OR null IS NULL) AND
  (p.age = '22' OR '22' IS NULL) AND
  true
```

Inline Type-Classes

THERE'S ACTUALLY A GOOD REASON TO DO THIS!


SO YOU SAY YOU CAN DO 'show'
TELL ME MORE ABOUT THIS TYPECLASS HIERARCHY OF YOURS
imgflip.com

An Inline Functor...

No, really. This is actually possible!

```
trait Functor[F[_]]:
  inline def map[A, B](inline xs: F[A], inline f: A => B): F[B]

class ListFunctor extends Functor[List]:
  inline def map[A, B](inline xs: List[A], inline f: A => B): List[B] = xs.map(f)

class QueryFunctor extends Functor[Query]:
  inline def map[A, B](inline xs: Query[A], inline f: A => B): Query[B] = xs.map(f)

inline given listFunctor as ListFunctor = new ListFunctor
inline given queryFunctor as QueryFunctor = new QueryFunctor

inline def doMap[F[_], A, B](inline from: F[A], inline f: A => B)(using inline fun: Functor[F]): F[B] =
  fun.map(from, f)
```

```
val list2 = doMap(List(1,2,3), (i: Int) => i + 1)                  // Runtime
inline def q = quote { doMap(select[Person], (p: Person) => p.name) }    // Compile-Time!
```

```
[info] Compiling 1 Scala source to /Users...
Compile Time Query Is: SELECT p.name FROM Person p

runMain miniquill.InlineMacroTest1FunctionalTypeclass
List(2, 3, 4)
```

TypeclassExample_FunctorOldStyle.scala

An Inline Functor… Dotty Idiomatically!

Tell them you saw it here first!

```
trait Functor[F[_]]:
  extension [A, B](inline x: F[A]):
    inline def map(inline f: A => B): F[B]

class ListFunctor extends Functor[List]:
  extension [A, B](inline xs: List[A])
    inline def map(inline f: A => B): List[B] = xs.map(f)

class QueryFunctor extends Functor[Query]:
  extension [A, B](inline xs: Query[A])
    inline def map(inline f: A => B): Query[B] = xs.map(f)

inline given listFunctor as ListFunctor = new ListFunctor
inline given queryFunctor as QueryFunctor = new QueryFunctor

extension [F[_], A, B](inline from: F[A])(using inline fun: Functor[F]):
  inline def mapF(inline f: A => B) = from.map(f)
```

```
println( List(1,2,3).mapF(i => i + 1) )
inline def q = quote { select[Person].mapF(p => p.name) }
```

TypeclassExample_Functor.scala

… and a Monad!

Well, not really a Monad since we don't have Pure.
Actually it's the FlatMap Typeclass

```
trait Monad[F[_]] extends Functor[F]:
  extension [A, B](inline x: F[A]):
    inline def map(inline f: A => B): F[B]
    inline def flatMap(inline f: A => F[B]): F[B]

class ListMonad extends Monad[List]:
  extension [A, B](inline xs: List[A])
    inline def map(inline f: A => B): List[B] = xs.map(f)
    inline def flatMap(inline f: A => List[B]): List[B] = xs.flatMap(f)

class QueryMonad extends Monad[Query]:
  extension [A, B](inline xs: Query[A])
    inline def map(inline f: A => B): Query[B] = xs.map(f)
    inline def flatMap(inline f: A => Query[B]): Query[B] = xs.flatMap(f)

inline given listMonad as ListMonad = new ListMonad
inline given queryMonad as QueryMonad = new QueryMonad

extension [F[_], A, B](inline from: F[A])(using inline fun: Monad[F]):
  inline def mapM(inline f: A => B) = from.map(f)
  inline def flatMapM(inline f: A => F[B]) = from.flatMap(f)
```

```
inline def q = quote { select[Person].mapF(p => p.name) }
inline def q = quote { select[Person].flatMapM(p => query[Address]) }
```

TypeclassExample_Monad.scala

Comprehend This!

... and you will be saved!

```scala
trait For[F[_]]:
  extension [A, B](inline x: F[A]):
    inline def map(inline f: A => B): F[B]
    inline def flatMap(inline f: A => F[B]): F[B]
    inline def withFilter(inline f: A => Boolean): F[A]

class ListFor extends For[List]:
  extension [A, B](inline xs: List[A])
    inline def map(inline f: A => B): List[B] = xs.map(f)
    inline def flatMap(inline f: A => List[B]): List[B] = xs.flatMap(f)
    inline def withFilter(inline f: A => Boolean): List[A] = xs.filter(f)

class QueryFor extends For[Query]:
  extension [A, B](inline xs: Query[A])
    inline def map(inline f: A => B): Query[B] = xs.map(f)
    inline def flatMap(inline f: A => Query[B]): Query[B] = xs.flatMap(f)
    inline def withFilter(inline f: A => Boolean): Query[A] = xs.withFilter(f)

inline given listFor as ListFor = new ListFor
inline given queryFor as QueryFor = new QueryFor
```

```scala
object UseCase:
  extension [F[_]](inline people: F[Person])(using inline fun: For[F]):
    inline def joesAddresses(inline addresses: F[Address]) =
      for {
        p <- people if (p.name == "Joe")
        a <- addresses if (p.id == a.fk)
      } yield (p, a)
```

TypeclassExample_For.scala

Comprehend This!

... and you will be saved!

```
object UseCase:
  extension [F[_]](inline people: F[Person])(using inline fun: For[F]):
    inline def joesAddresses(inline addresses: F[Address]) =
      for {
        p <- people if (p.name == "Joe")
        a <- addresses if (p.id == a.fk)
      } yield (p, a)
```

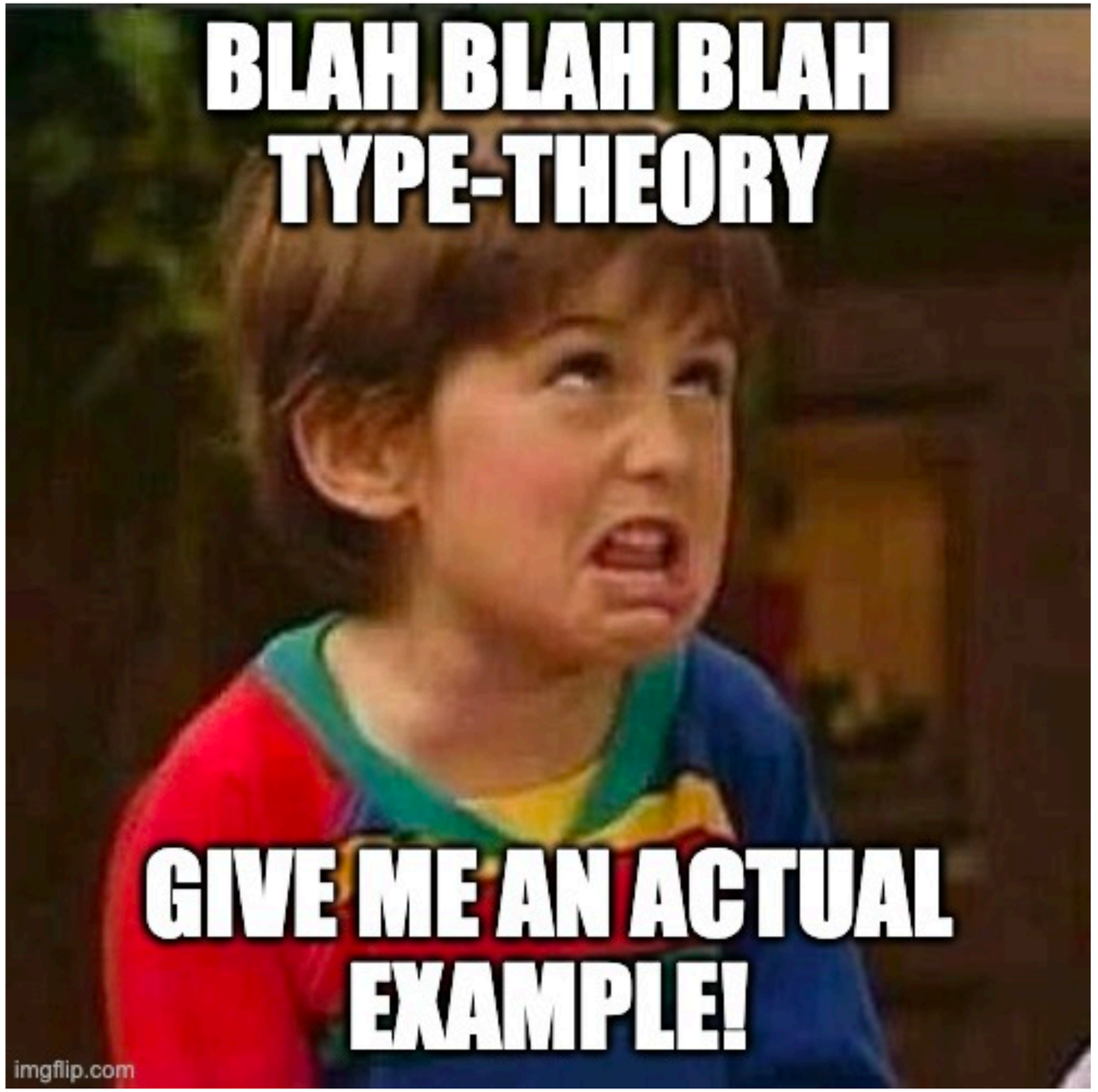
Compile Time

```
inline def q = quote { people.joesAddresses(addresses) }

[info] Compiling 1 Scala source to /Users…
SELECT p.id, p.name, p.age, a.fk, a.street, a.zip FROM Person p, Address a
WHERE p.name = 'Joe' AND p.id = a.fk
```

Runtime

```
val peopleL = List(Person(1, "Joe", 22), Person(2, "Jack", 33), Person(3, "James", 44))
val addressesL = List(Address(1, "123 St.", 111), Address(2, "456 St.", 222), Address(3, "789 St.", 333))

println(peopleL.joesAddresses(addressesL))

[info] running miniquill.InlineMacroTest1FunctionalTypeclassIdiomatic
List((Person(1,Joe,22),Address(1,123 St.,111)))
```

TypeclassExample_For.scala


BLAH BLAH BLAH
TYPE-THEORY
GIVE ME AN ACTUAL
EXAMPLE!
imgflip.com

A Practical Example

... you say boring, I say believable!

Node

Log
Time1 - Node1 Update
Time2 - Node2 Update
Time3 - Node1 Update
Time4 - Node1 Update
Time5 - Node2 Update

Master

Log
Time1 - Master1 Update
Time2 - Master2 Update
Time3 - Master3 Update
Time4 - Master1 Update
Time5 - Master3 Update

Worker

Log
Time1 - Node1 Update
Time2 - Node3 Update
Time3 - Node3 Update
Time4 - Node2 Update
Time5 - Node2 Update

A Practical Example

... you say boring, I say believable!

Node		
id	**timestamp**	**status**
1	1	UP
2	2	UP
1	3	DOWN
1	4	UP
2	5	DOWN

Latest Node Status		
id	**timestamp**	**status**
1	4	UP
2	5	DOWN

Master		
id	**lastCheck**	**state**
1	1	IDLE
2	2	WAITING
3	3	WAITING
1	4	WAITING
3	5	FINISHED

Latest Master Status		
id	**lastCheck**	**state**
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Worker		
shard	**lastTime**	**reply**
1	1	BUSY
3	2	BUSY
3	3	DONE
2	4	DONE
2	5	BUSY

Latest Worker Status		
shard	**lastTime**	**reply**
1	5	BUSY
2	2	BUSY
3	5	DONE

A Practical Example

... you say boring, I say believable!

```
SELECT
  a.id, a.timestamp, a.status
FROM Node a
LEFT JOIN Node b
ON b.id = a.id
AND b.timestamp > a.timestamp
WHERE b.id IS NULL
```

```
SELECT
  a.key, a.lastCheck, a.state
FROM Master a
LEFT JOIN Master b
ON b.key = a.key
AND b.lastCheck > a.lastCheck
WHERE b.key IS NULL
```

```
SELECT
  a.shard, a.lastTime, a.reply
FROM Worker a
LEFT JOIN Worker b
ON b.shard = a.shard
AND b.lastTime > a.lastTime
WHERE b.shard IS NULL
```

Latest Node Status

id	timestamp	status
1	4	UP
2	5	DOWN

Latest Master Status

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

TypeclassUsecase_Repetitive.scala

A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {
  query[Node].leftJoin(query[Node])
  .on((a, b) =>
      b.id == a.id &&
      b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

```
inline def masters = quote {
  query[Master].leftJoin(query[Master])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

```
inline def workers = quote {
  query[Worker].leftJoin(query[Worker])
  .on((a, b) =>
      b.shard == a.shard &&
      b.lastTime > a.lastTime
  )
  .filter((a, b) =>
    b.map(_.shard).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Status

id	timestamp	status
1	4	UP
2	5	DOWN

Latest Master Status

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {
  query[Node].leftJoin(query[Node])
  .on((a, b) =>
      b.id == a.id &&
      b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Status

id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def masters = quote {
  query[Master].leftJoin(query[Master])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Status

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

```
inline def masters = quote {
  query[Master].leftJoin(query[Master])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Status

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def masters = quote {
  query[Entity].leftJoin(query[Entity])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Entity

id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def workers = quote {
  query[Service].leftJoin(query[Service])
  .on((a, b) =>
      b.shard == a.shard &&
      b.lastTime > a.lastTime
  )
  .filter((a, b) =>
    b.map(_.shard).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Service

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

```
inline def masters = quote {
  query[Switch].leftJoin(query[Switch])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Switch

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def nodes = quote {
  query[App].leftJoin(query[App])
  .on((a, b) =>
      b.id == a.id &&
      b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node App

id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def masters = quote {
  query[Bridge].leftJoin(query[Bridge])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Bridge

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

```
inline def masters = quote {
  query[Combo].leftJoin(query[Combo])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Combo

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def masters = quote {
  query[Reso].leftJoin(query[Reso])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Reso

id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def workers = quote {
  query[Database].leftJoin(query[Database])
  .on((a, b) =>
      b.shard == a.shard &&
      b.lastTime > a.lastTime
  )
  .filter((a, b) =>
    b.map(_.shard).isEmpty)
  .map((a, b) => a)
}
```

Latest Master Database

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

```
inline def masters = quote {
  query[Route].leftJoin(query[Route])
  .on((a, b) =>
      b.key == a.key &&
      b.lastCheck > a.lastCheck
  )
  .filter((a, b) =>
    b.map(_.key).isEmpty)
  .map((a, b) => a)
}
```

Latest Worker Route

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

A Practical Example

... you say boring, I say believable!

```
inline def nodes = quote {
  query[Node].leftJoin(query[Nod
  .on((a, b) =>
      b.id == a.id &&
      b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node Status

id	timestamp	status
1	4	UP
2	5	DOWN

```
inline def nodes = quote {
  query[App].leftJoin(query[App]
  .on((a, b) =>
      b.id == a.id &&
      b.timestamp > a.timestamp
  )
  .filter((a, b) =>
    b.map(_.id).isEmpty)
  .map((a, b) => a)
}
```

Latest Node App

id	timestamp	status
1	4	UP
2	5	DOWN

3	5	FINISHED

3	5	DONE

2	5	DOWN

3	5	FINISHED

```
ine def masters = quote {
uery[Switch].leftJoin(query[Switch])
on((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck

filter((a, b) =>
 b.map(_.key).isEmpty)
map((a, b) => a)
```

st Worker Switch

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
ne def masters = quote {
ery[Route].leftJoin(query[Route])
n((a, b) =>
    b.key == a.key &&
    b.lastCheck > a.lastCheck

ilter((a, b) =>
 b.map(_.key).isEmpty)
ap((a, b) => a)
```

st Worker Route

shard	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

A Practical Example

... you say boring, I say believable!

```
inline def latestStatus[T, G](
  inline q: Query[T])(
  inline groupKey: T => G,
  inline earlierThan: (T, T) => Boolean
): Query[T] =
  q.leftJoin(q)
  .on((a, b) =>
      groupKey(b) == groupKey(a) &&
      earlierThan(b, a)
  )
  .filter((a, b) =>
    b.map(b => groupKey(b)).isEmpty)
  .map((a, b) => a)
```

Latest Node Status

id	timestam	status
1	4	UP
2	5	DOWN

Latest Master Status

id	lastCheck	state
1	5	WAITING
2	2	WAITING
3	5	FINISHED

Latest Worker Status

id	lastTime	reply
1	5	BUSY
2	2	BUSY
3	5	DONE

```
inline def nodesLatest = quote {
  latestStatus(query[Node])(
    n => n.id,
    (a, b) => a.timestamp < b.timestamp)
}
```

```
inline def mastersLatest = quote {
  latestStatus(query[Master])(
    m => m.key,
    (a, b) => a.lastCheck < b.lastCheck)
}
```

```
inline def workersLatest = quote {
   latestStatus(query[Worker])(
     w => w.shard,
     (a, b) => a.lastTime < b.lastTime)
 }
```

TypeclassUsecase_Encapsulated.scala

A Practical Example

... you say boring, I say believable!

```
trait GroupKey[T, G]:
  inline def apply(inline t: T): G
trait EarlierThan[T]:
  inline def apply(inline a: T, inline b: T): Boolean
```

TypeclassUsecase_Typeclass.scala

```scala
class NodeGroupKey extends GroupKey[Node, Int]:
  inline def apply(inline t: Node): Int = t.id
class MasterGroupKey extends GroupKey[Master, Int]:
  inline def apply(inline t: Master): Int = t.key
class WorkerGroupKey extends GroupKey[Worker, Int]:
  inline def apply(inline t: Worker): Int = t.shard

inline given nodeGroupKey as NodeGroupKey = new NodeGroupKey
inline given workerGroupKey as WorkerGroupKey = new WorkerGroupKey
inline given masterGroupKey as MasterGroupKey = new MasterGroupKey
```

```scala
class NodeEarlierThan extends EarlierThan[Node]:
  inline def apply(inline a: Node, inline b: Node) = a.timestamp < b.timestamp
class MasterEarlierThan extends EarlierThan[Master]:
  inline def apply(inline a: Master, inline b: Master) = a.lastCheck < b.lastCheck
class WorkerEarlierThan extends EarlierThan[Worker]:
  inline def apply(inline a: Worker, inline b: Worker) = a.lastTime < b.lastTime

inline given nodeEarlierThan as NodeEarlierThan = new NodeEarlierThan
inline given workerEarlierThan as WorkerEarlierThan = new WorkerEarlierThan
inline given masterEarlierThan as MasterEarlierThan = new MasterEarlierThan
```

TypeclassUsecase_Typeclass.scala

```scala
inline def latestStatus[T, G](
  inline q: Query[T])(
  using inline groupKey: GroupKey[T, G],
  inline earlierThan: EarlierThan[T]
): Query[T] =
  q.leftJoin(q)
  .on((a, b) =>
      groupKey(b) == groupKey(a) &&
      earlierThan(b, a)
  )
  .filter((a, b) =>
    b.map(b => groupKey(b)).isEmpty)
  .map((a, b) => a)
```

```scala
quote { latestStatus(query[Node]) }
quote { latestStatus(query[Master]) }
quote { latestStatus(query[Worker]) }
```

TypeclassUsecase_Typeclass.scala

```scala
inline def latestStatus[T, G](
  inline q: Query[T])(
  using inline groupKey: GroupKey[T, G],
  inline earlierThan: EarlierThan[T]
): Query[T] =
  q.leftJoin(q)
  .on((a, b) =>
      groupKey(b) == groupKey(a) &&
      earlierThan(b, a)
  )
  .filter((a, b) =>
    b.map(b => groupKey(b)).isEmpty)
  .map((a, b) => a)
```

```scala
quote { latestStatus(query[Node]) }
```

```
SELECT
  a.id, a.timestamp, a.status
FROM Node a
LEFT JOIN Node b
ON b.id = a.id
AND b.timestamp > a.timestamp
WHERE b.id IS NULL
```

```scala
quote { latestStatus(query[Master]) }
```

```
SELECT
  a.key, a.lastCheck, a.state
FROM Master a
LEFT JOIN Master b
ON b.key = a.key
AND b.lastCheck > a.lastCheck
WHERE b.key IS NULL
```

```scala
quote { latestStatus(query[Worker]) }
```

```
SELECT
  a.shard, a.lastTime, a.reply
FROM Worker a
LEFT JOIN Worker b
ON b.shard = a.shard
AND b.lastTime > a.lastTime
WHERE b.shard IS NULL
```

TypeclassUsecase_Typeclass.scala


```
quote { latestStatus(query[Node]) }
```

```
  SELECT
    a.id, a.timestamp, a.status
  FROM Node a
  LEFT JOIN Node b
  ON b.id = a.id
  AND b.timestamp > a.timestamp
  WHERE b.id IS NULL
```

```
quote { latestStatus(query[Master]) }
```

```
  SELECT
    a.key, a.lastCheck, a.state
  FROM Master a
  LEFT JOIN Master b
  ON b.key = a.key
  AND b.lastCheck > a.lastCheck
  WHERE b.key IS NULL
```

```
quote { latestStatus(query[Worker]) }
```

```
  SELECT
    a.shard, a.lastTime, a.reply
  FROM Worker a
  LEFT JOIN Worker b
  ON b.shard = a.shard
  AND b.lastTime > a.lastTime
  WHERE b.shard IS NULL
```

TypeclassUsecase_Typeclass.scala

Let's Take One More Step

... because we can!

```scala
trait JoiningFunctor[F[_]]:
  extension [A, B](inline xs: F[A])
    inline def map(inline f: A => B): F[B]
    inline def filter(inline f: A => Boolean): F[A]
    inline def leftJoin(inline ys: F[B])(inline f: (A, B) => Boolean): F[(A, Option[B])]
```

TypeclassUsecase_TypeclassWithList.scala

Now Implement!

Faster! Faster!

```
class QueryJoiningFunctor extends JoiningFunctor[Query]:
  extension [A, B](inline xs: Query[A])
    inline def map(inline f: A => B): Query[B] = xs.map(f)
    inline def filter(inline f: A => Boolean): Query[A] = xs.filter(f)
    inline def leftJoin(inline ys: Query[B])(inline f: (A, B) => Boolean): Query[(A, Option[B])] =
      xs.leftJoin(ys).on(f)

class ListJoiningFunctor extends JoiningFunctor[List]:
  extension [A, B](inline xs: List[A])
    inline def map(inline f: A => B): List[B] = xs.map(f)
    inline def filter(inline f: A => Boolean): List[A] = xs.filter(f)
    inline def leftJoin(inline ys: List[B])(inline f: (A, B) => Boolean): List[(A, Option[B])] =
      xs.flatMap { x =>
        val matching = ys.filter(y => f(x, y)).map(y => (x, Some(y)))
        if (matching.length == 0) List((x, None)) else matching
      }

inline given queryJoiningFunctor as QueryJoiningFunctor = new QueryJoiningFunctor
inline given listJoiningFunctor as ListJoiningFunctor = new ListJoiningFunctor
```

TypeclassUsecase_TypeclassWithList.scala

With a Query! With a List!

Believe me now?

```scala
inline def latestStatus[F[_], T, G](inline q: F[T])(
  using inline fun: JoiningFunctor[F],
  inline groupKey: GroupKey[T, G],
  inline earlierThan: EarlierThan[T]): F[T] =
    q.leftJoin(q)((a, b) =>
        groupKey(b) == groupKey(a) &&
        earlierThan(b, a)
    )
    .filter((a, b) =>
      b.map(b => groupKey(b)).isEmpty)
    .map((a, b) => a)
```

```scala
quote {
  latestStatus(select[Node])
}
```

```
SELECT
  a.id, a.timestamp, a.status
FROM Node a
LEFT JOIN Node b
ON b.id = a.id
AND b.timestamp > a.timestamp
WHERE b.id IS NULL
```

```scala
val nodesList = List(
  Node(1, 1, "UP"),
  Node(1, 2, "DOWN"),
  Node(2, 3, “WAITING")
)
println( latestStatus(nodesList) )
```

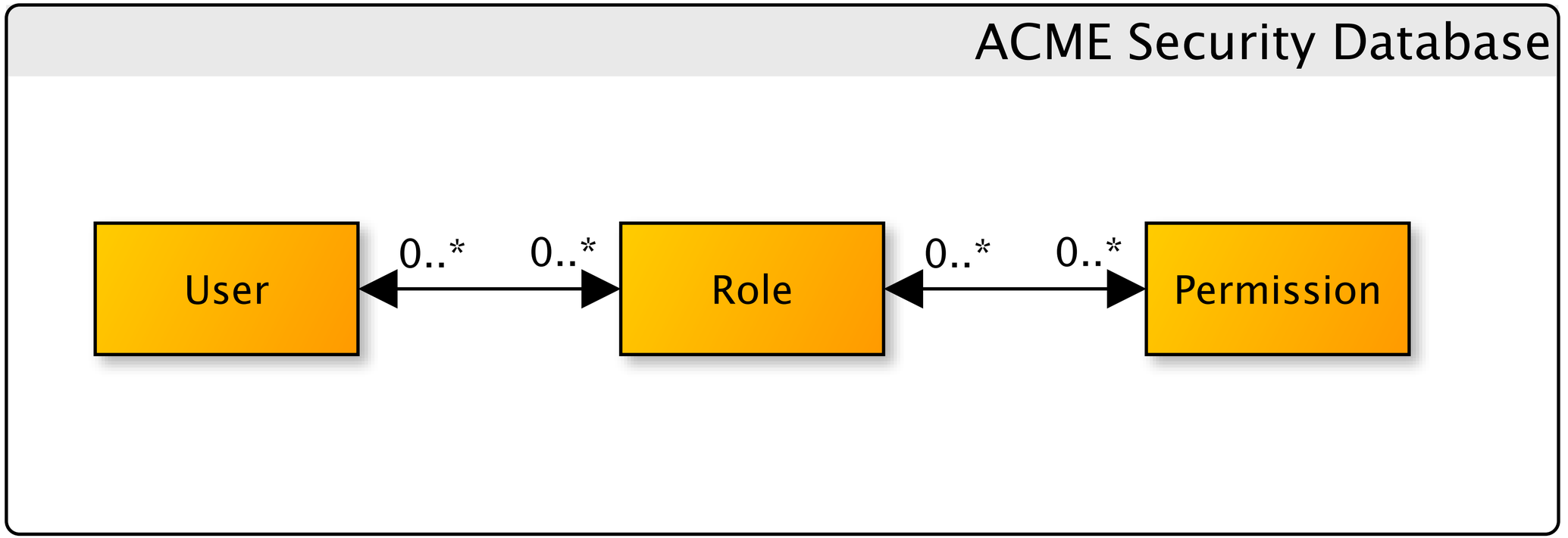
```
List(Node(1,1,UP), Node(2,3,WAITING))
```

TypeclassUsecase_TypeclassWithList.scala

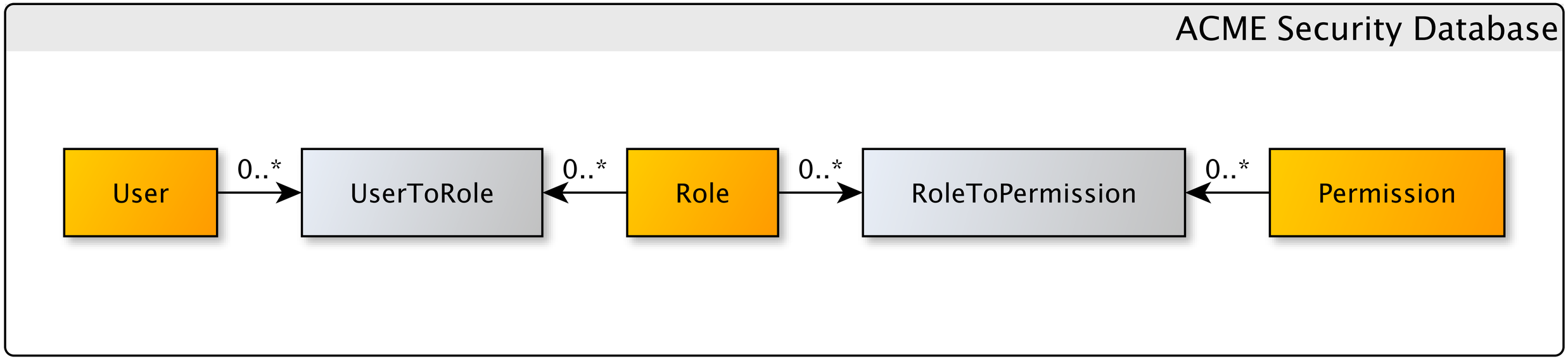
Inline Type-Level Programming

SERIOUSLY…WE’RE DOING THIS!

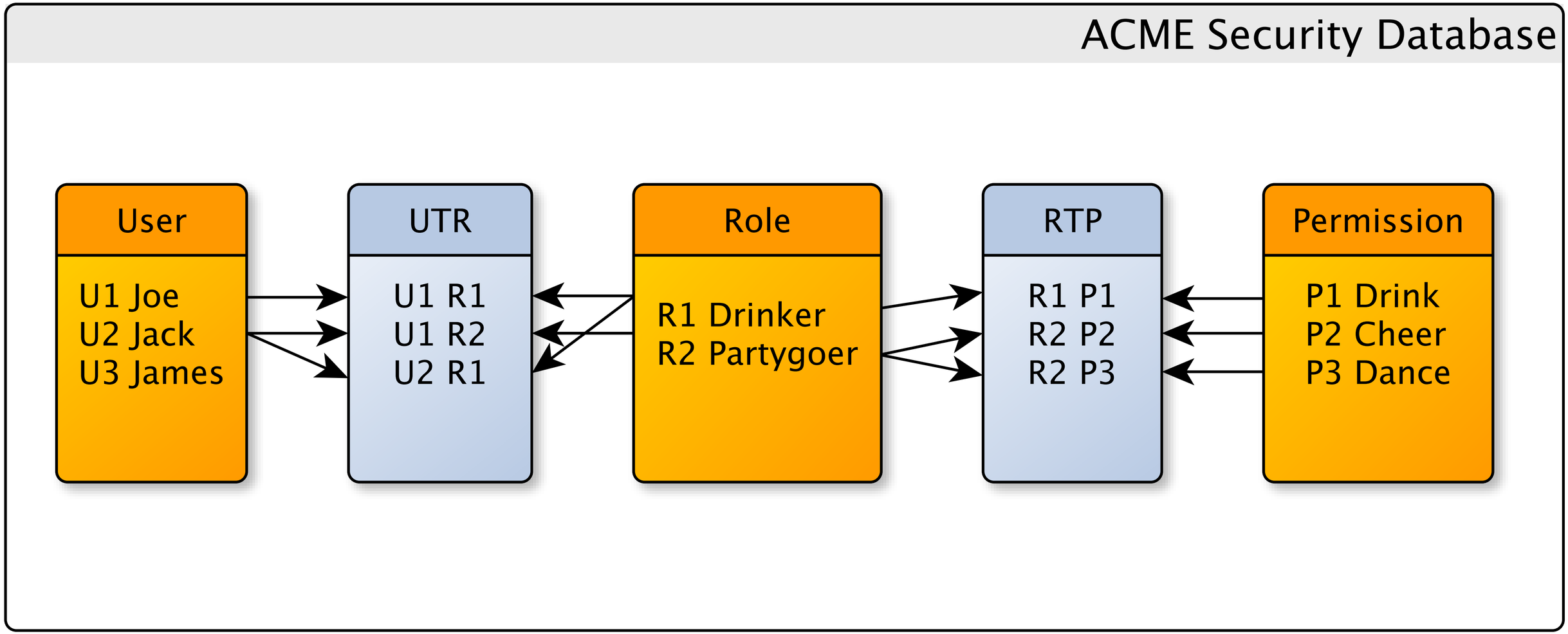
Every Security DB in the Universe…

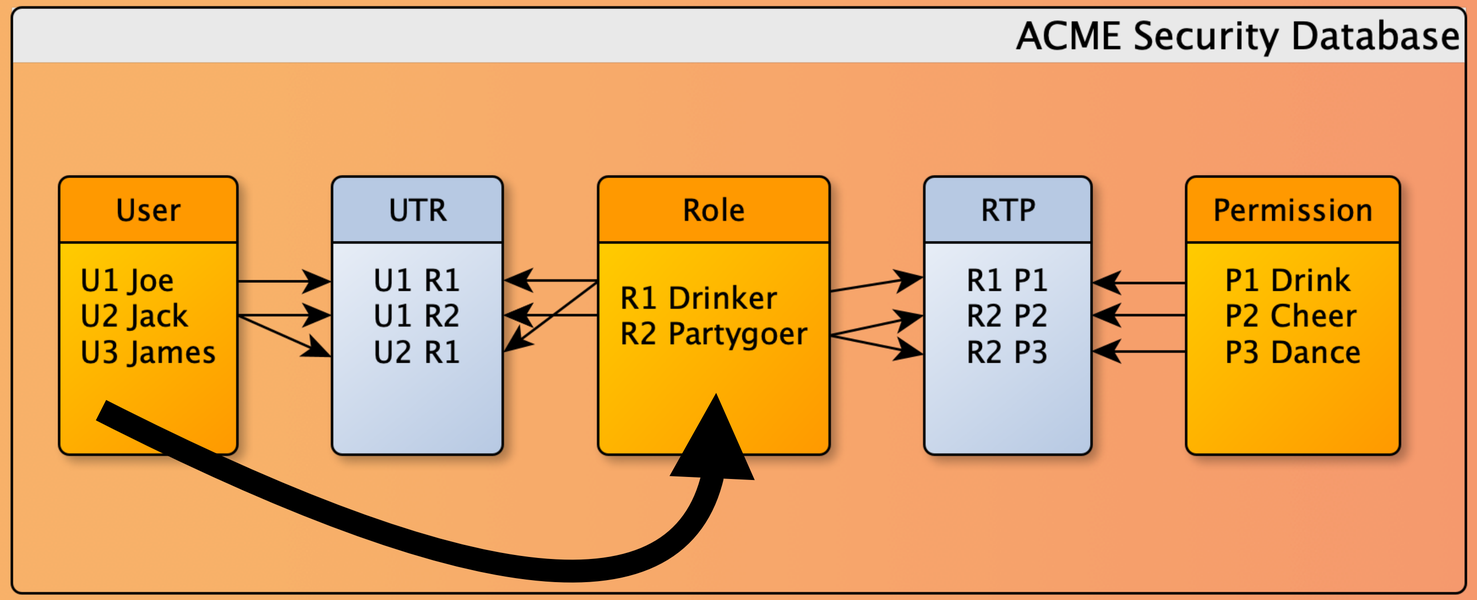


Actually it’s…

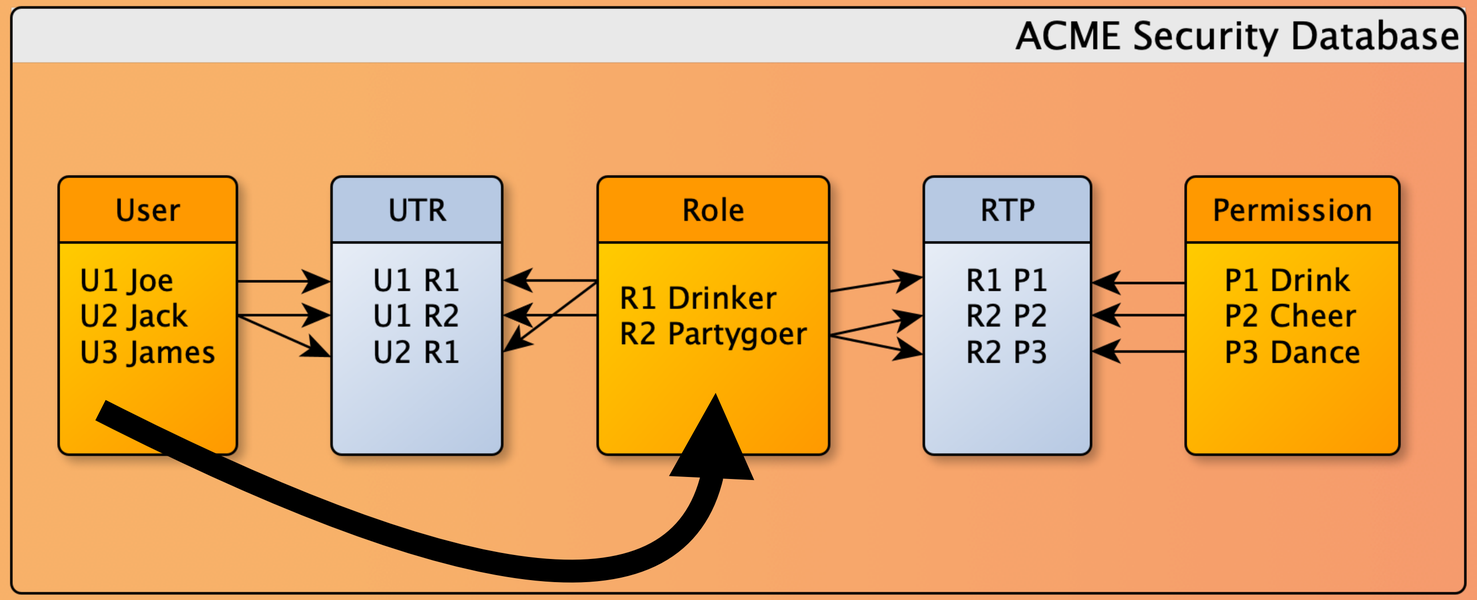


Actually it's…



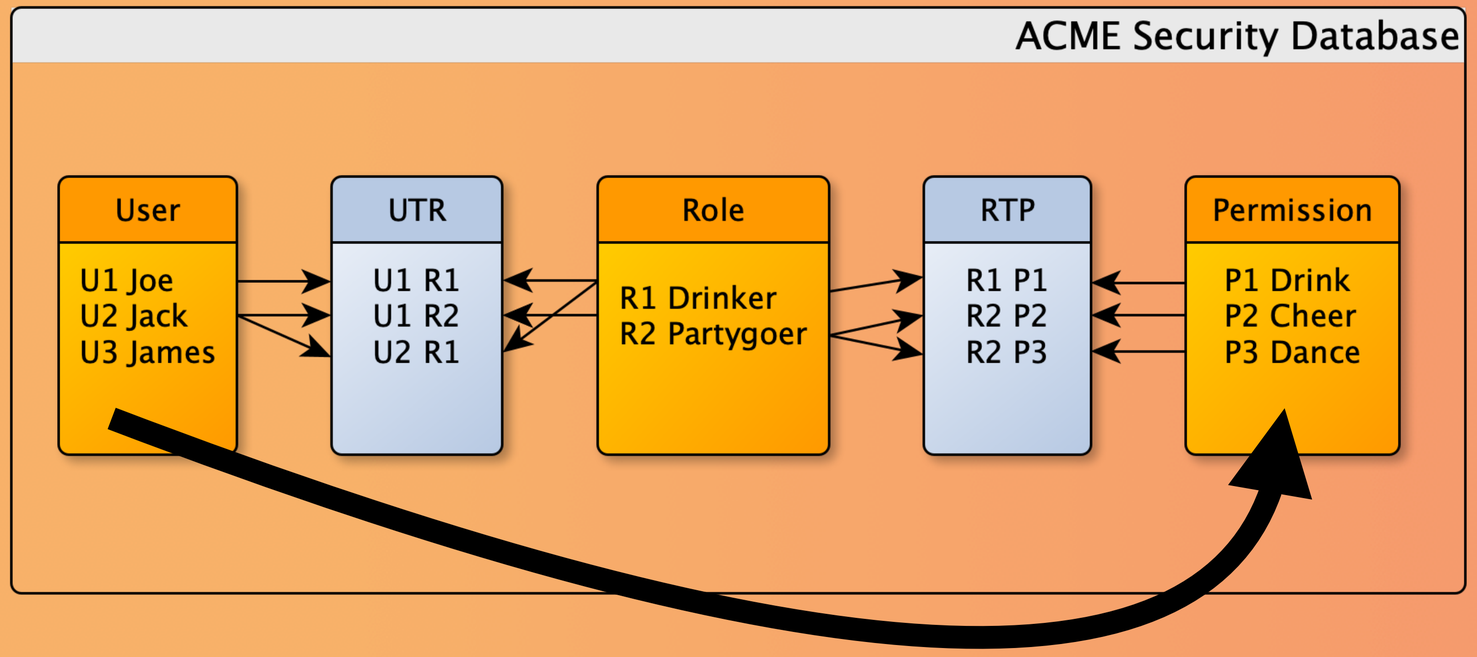



```
SELECT
  s.id, s.name,
  r.id, r.name
FROM
  User s
  INNER JOIN UserToRole so ON so.userId = s.id
  INNER JOIN Role r ON r.id = so.roleId
```

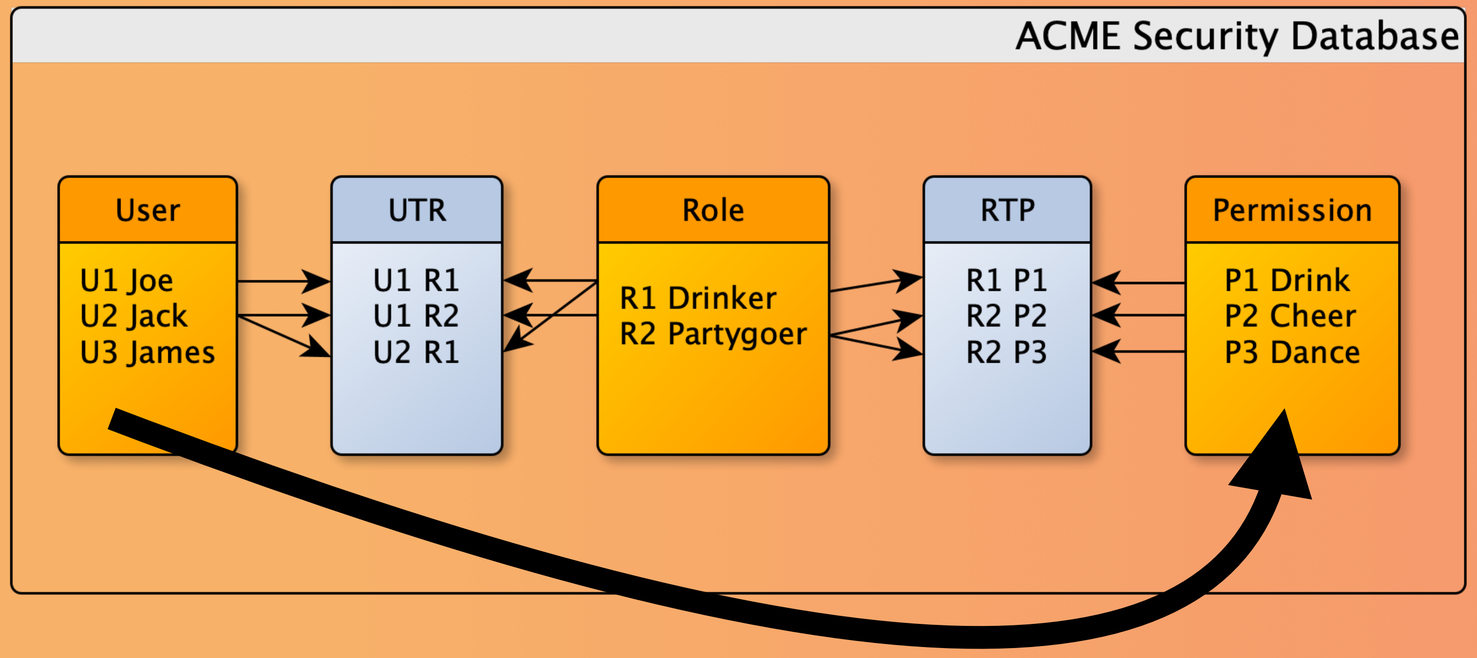



```
SELECT
  s.id, s.name,
  r.id, r.name
FROM
  User s
  INNER JOIN UserToRole so ON so.userId = s.id
  INNER JOIN Role r ON r.id = so.roleId
```

```
for {
  s <- query[User]
  sr <- query[UserToRole].join(sr => sr.userId == s.id)
  r <- query[Role].join(r => r.id == sr.roleId)
} yield (s, r)
```



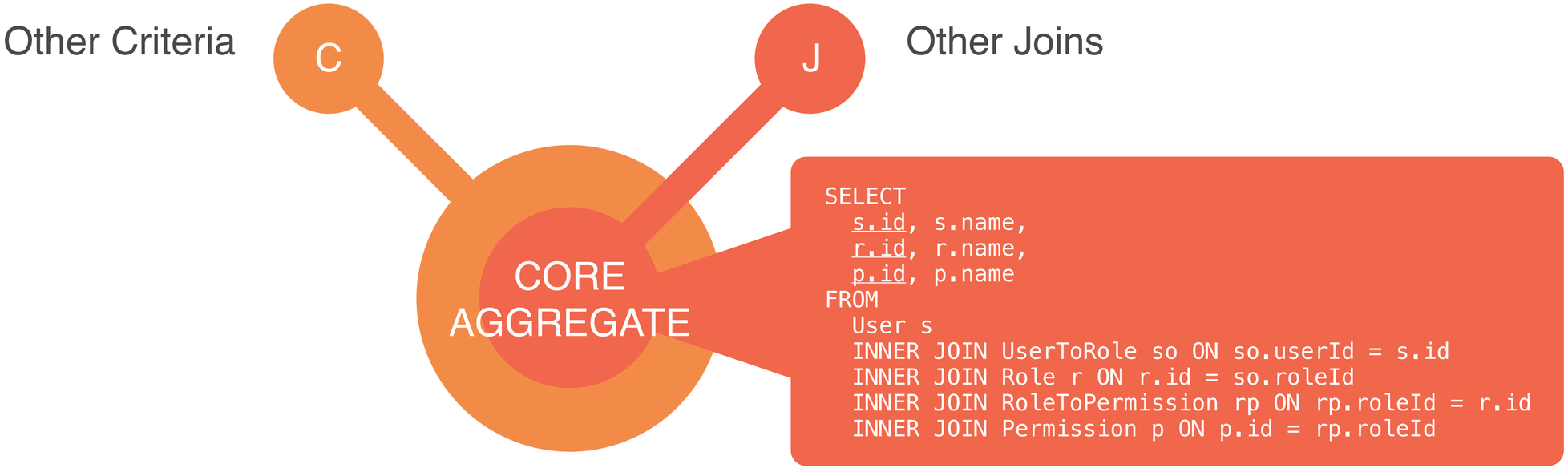

```
SELECT
  s.id, s.name,
  r.id, r.name,
  p.id, p.name
FROM
  User s
  INNER JOIN UserToRole so ON so.userId = s.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
```

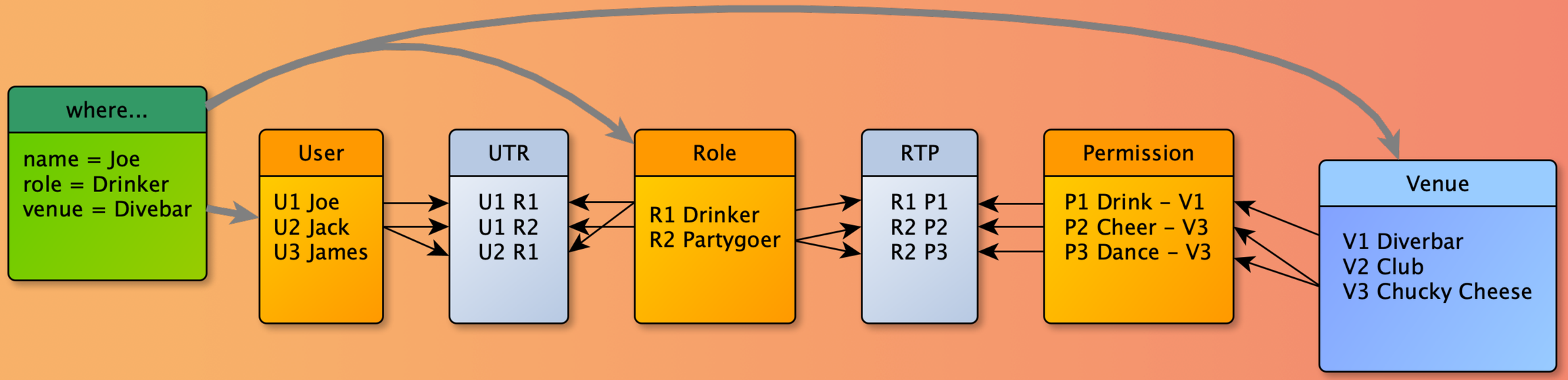



```
SELECT
  s.id, s.name,
  r.id, r.name,
  p.id, p.name
FROM
  User s
  INNER JOIN UserToRole so ON so.userId = s.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
```

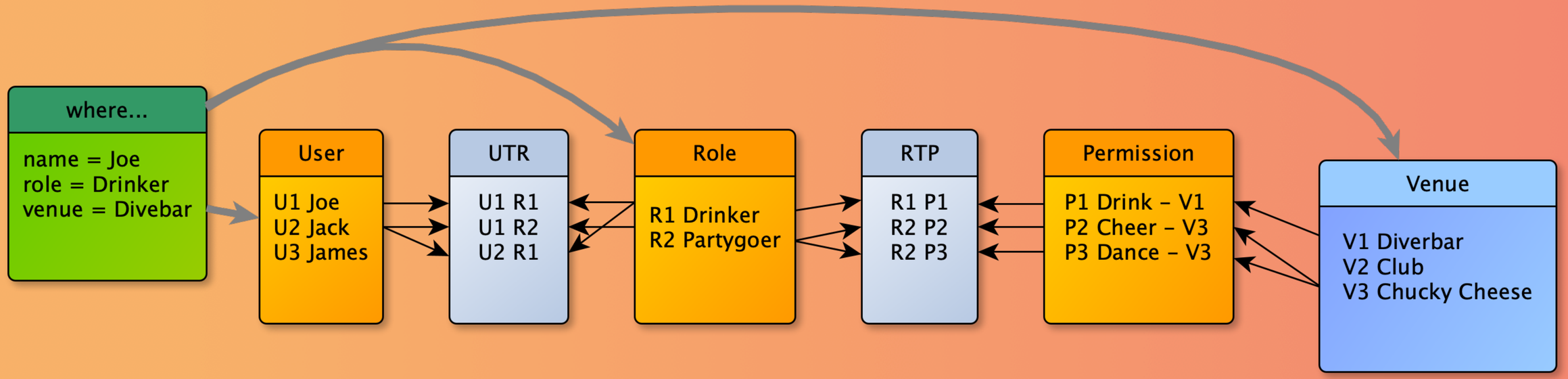
```
for {
  u  <- query[User]
  ur <- query[UserToRole].join(so => so.userId == u.id)
  r  <- query[Role].join(r => r.id == ur.roleId)
  rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
  p  <- query[Permission].join(p => p.id == rp.roleId)
} yield (u, r, p)
```

… but usually, there's other stuff






```
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```




```
for {
  u  <- query[User] if (u.name == "Joe")
  ur <- query[UserToRole].join(so => so.userId == u.id)
  r  <- query[Role].join(r => r.id == ur.roleId) if (r.name == "Drinker")
  rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
  p  <- query[Permission].join(p => p.id == rp.roleId)
  v  <- query[Venue].join(v => v.id == p.venueId) if (v.name == "Divebar")
} yield (u, r, p, v)
```


```
for {
  u  <- query[User] if (u.name == "Joe")
  ur <- query[UserToRole].join(so => so.userId == u.id)
  r  <- query[Role].join(r => r.id == ur.roleId) if (r.name == "Drinker")
  rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
  p  <- query[Permission].join(p => p.id == rp.roleId)
  v  <- query[Venue].join(v => v.id == p.venueId) if (v.name == "Divebar")
} yield (u, r, p, v)
```

Bring on the API Calls!...

Sanity Not Included

```
val userAndRole = quote {
  for {
    u  <- query[User]
    ur <- query[UserToRole].join(so => so.userId == u.id)
    r  <- query[Role].join(r => r.id == ur.roleId)
 } yield (u, r, p)
}
```

inline def userAndRole =
quote { ... } => Query[(User,Role)]

```
val userAndRoleAndPermission = quote {
  for {
    u  <- query[User]
    ur <- query[UserToRole].join(so => so.userId == u.id)
    r  <- query[Role].join(r => r.id == ur.roleId)
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
    p  <- query[Permission].join(p => p.id == rp.roleId)
 } yield (u, r, p)
}
```

inline def userAndRoleAndPermission =
quote { ... } => Query[(User,Role,Permission)]

```
val userAndRoleAndPermissionAndVenue = quote {
  for {
    u  <- query[User]
    ur <- query[UserToRole].join(so => so.userId == u.id)
    r  <- query[Role].join(r => r.id == ur.roleId)
    rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
    p  <- query[Permission].join(p => p.id == rp.roleId)
    v  <- query[Venue].join(v => v.id == p.venueId)
 } yield (u, r, p, v)
}
```

inline def userAndRoleAndPermissionAndVenue =
quote { ... } => Q[(User,Role,Permission,Venue)]

Is there a better way
to do this???

WHEN THE ALTERNATIVE IS PURE MADNESS...

Type-Level It!

WHEN THE ALTERNATIVE IS PURE MADNESS...

Type-Level It!

```
trait Path[From, To]:
  type Out
  inline def get: Out
```

Magic Is Here!

TypelevelUsecase.scala

WHEN ALL ELSE FAILS...

Type - Level It!

```
trait Path[User, Role]:
  type Out = Query[(User, Role)]

trait Path[User, Permission]:
  type Out = Query[(User, Role, Permission)]

trait Path[User, Venue]:
  type Out = Query[(User, Role, Permission, Venue)]
```

TypelevelUsecase.scala

```scala
class PathFromUserToRole extends Path[User, Role]:
  type Out = Query[(User, Role)]
  inline def get: Query[(User, Role)] =
    for {
      u <- query[User]
      ur <- query[UserToRole].join(ur => ur.userId == s.id)
      r <- query[Role].join(r => r.id == ur.roleId)
    } yield (u, r)

class PathFromUserToPermission extends Path[User, Permission]:
  type Out = Query[(User, Role, Permission)]
  inline def get: Query[(User, Role, Permission)] =
    for {
      u <- query[User]
      ur <- query[UserToRole].join(ur => ur.userId == s.id)
      r <- query[Role].join(r => r.id == ur.roleId)
      rp <- query[RoleToPermission].join(rp => rp.roleId == r.id)
      p <- query[Permission].join(p => p.id == rp.roleId)
    } yield (u, r, p)

inline given pathFromUserToRole as PathFromUserToRole = new PathFromUserToRole
inline given pathFromUserToPermission as PathFromUserToPermission = new PathFromUserToPermission


inline def path[F, T](using inline path: Path[F, T]): path.Out = path.get
```

TypelevelUsecase.scala

WHEN ALL ELSE FAILS...

Type - Level It!

```
inline def q: Quoted[Query[(User, Role)]] =
  quote {
    path[User,Role]
  }

inline def q: Quoted[Query[(User, Role, Permission)]] =
  quote {
    path[User,Permission]
  }

inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =
  quote {
    path[User,Venue]
  }
```

TypelevelUsecase.scala

```scala
inline def q: Quoted[Query[(User, Role)]] =
  quote {
    path[User,Role]
  }
```

```scala
inline def q: Quoted[Query[(User, Role, Permission)]] =
  quote {
    path[User,Permission]
  }
```

```scala
inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =
  quote {
    path[User,Venue]
  }
```

TypelevelUsecase.scala

```scala
 inline def q: Quoted[Query[(User, Role)]] =
   quote {
     path[User,Role]
       .filter(t => t._1.name == "Joe" && t._2.name == "Drinker")
   }

inline def q: Quoted[Query[(User, Role, Permission)]] =
  quote {
    path[User,Permission]
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink")
  }

inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =
  quote {
    path[User,Venue]
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink"
         && t._4.name == "Divebar")
  }
```

TypelevelUsecase.scala

```scala
inline def q: Quoted[Query[(User, Role, Permission, Venue)]] =
  quote {
    path[User,Venue]
      .filter(t => t._1.name == "Joe" && t._2.name == "Drinker" && t._3.name == "Drink"
         && t._4.name == "Divebar")
  }
```

```sql
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

TypelevelUsecase.scala

```scala
inline def q =
  quote {
    path[User,Venue].filter { case (u, r, p, v) =>
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"
    }
  }
```

```
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

TypelevelUsecase.scala

```scala
inline def q =
  quote {
    path[User,Venue].filter((u, r, p, v) =>
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"
    )
  }
```

```
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

TypelevelUsecase.scala

```scala
inline def q =
  quote {
    path[User,Venue].filter((u, r, p, v) =>
      u.name == "Joe" && r.name == "Drinker" && p.name == "Drink" && v.name == "Divebar"
    )
  }

run(q)
```

```
sbt:dotty-simple> compile
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
[success] Total time: 7 s, completed Nov 23, 2020 12:16:08 AM
```

TypelevelUsecase.scala

```scala
inline def q =
  quote {
    path[User,Role]
      .filter((u, r) =>
        u.name == "Joe" &&
        r.name == "Drinker")
  }
```

```sql
SELECT
  u.id,  u.name, r.id, r.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker'
```

```scala
inline def q = quote {
    path[User,Permission]
      .filter((u, r, p) =>
        u.name == "Joe" &&
        r.name == "Drinker" &&
        p.name == "Drink")
  }
```

```sql
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink'
```

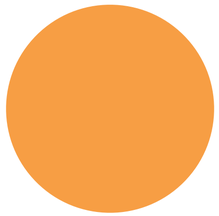
```scala
inline def q = quote {
    path[User,Venue]
      .filter((u, r, p, v) =>
        u.name == "Joe" &&
        r.name == "Drinker" &&
        p.name == "Drink" &&
        v.name == "Divebar")
  }
```

```sql
SELECT
  u.id,  u.name, r.id, r.name, p.id, p.name, p.venueId, v.id, v.name
FROM
  User u
  INNER JOIN UserToRole so ON so.userId = u.id
  INNER JOIN Role r ON r.id = so.roleId
  INNER JOIN RoleToPermission rp ON rp.roleId = r.id
  INNER JOIN Permission p ON p.id = rp.roleId
  INNER JOIN Venue v ON v.id = p.venueId
WHERE
  u.name = 'Joe' AND r.name = 'Drinker' AND p.name = 'Drink' AND v.name = 'Divebar'
```

TypelevelUsecase.scala

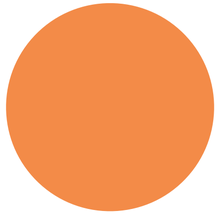

How Far...
Is just that little bit over there?

Conclusions



Inline is the new Implicit

In a good way! It will reveal many new applications in the years to come.



Dotty Enables Quill to be Awesome!

Many new capabilities we've only dreamed about up to now!