dave gurnell

untyped

myna

mynaweb.com

A Route to the Three 'R's: Reading, Writing, and the REST

https://github.com/davegurnell/scalalol-2011-talk

/add/123/to/234

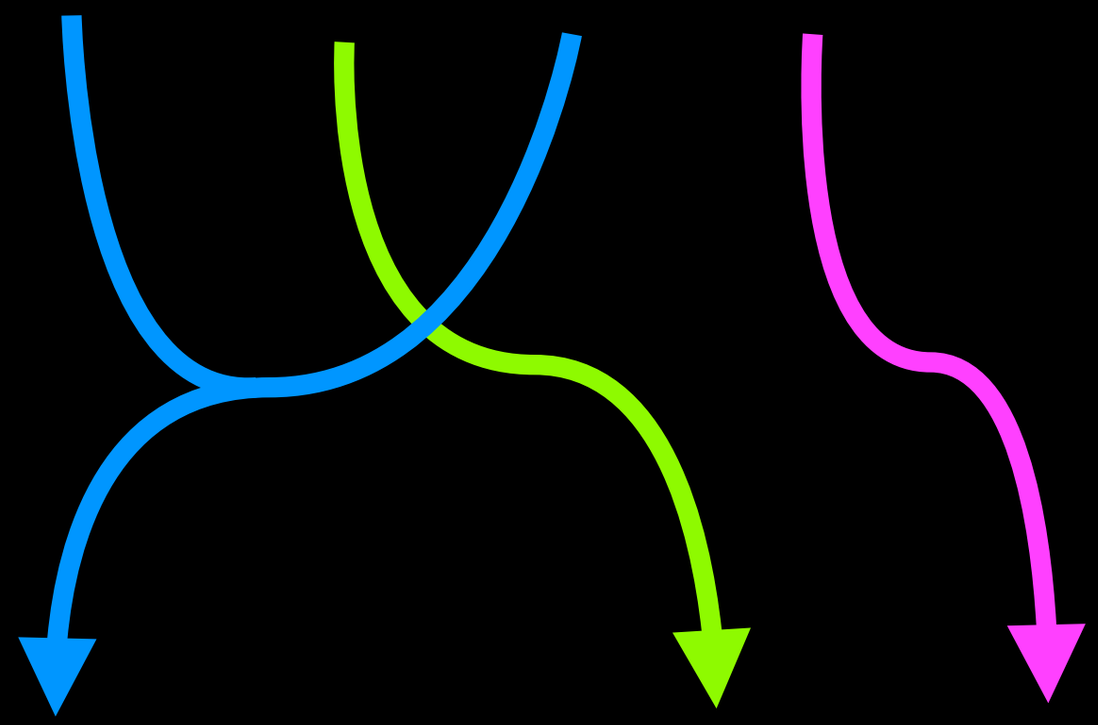
```
/add/123/to/234
```

```
Calculator.add(123, 234)
```

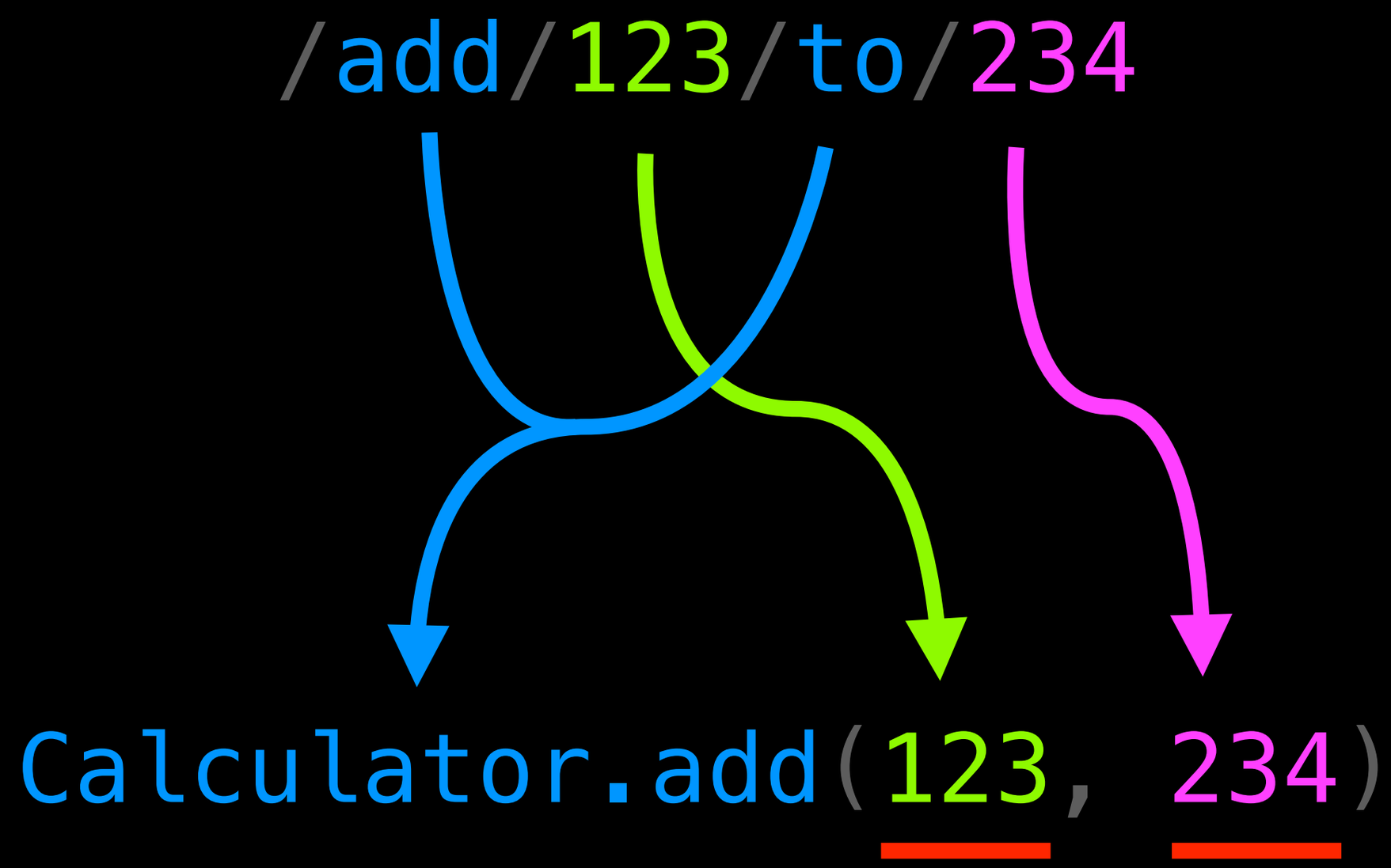
```
/add/123/to/234
```

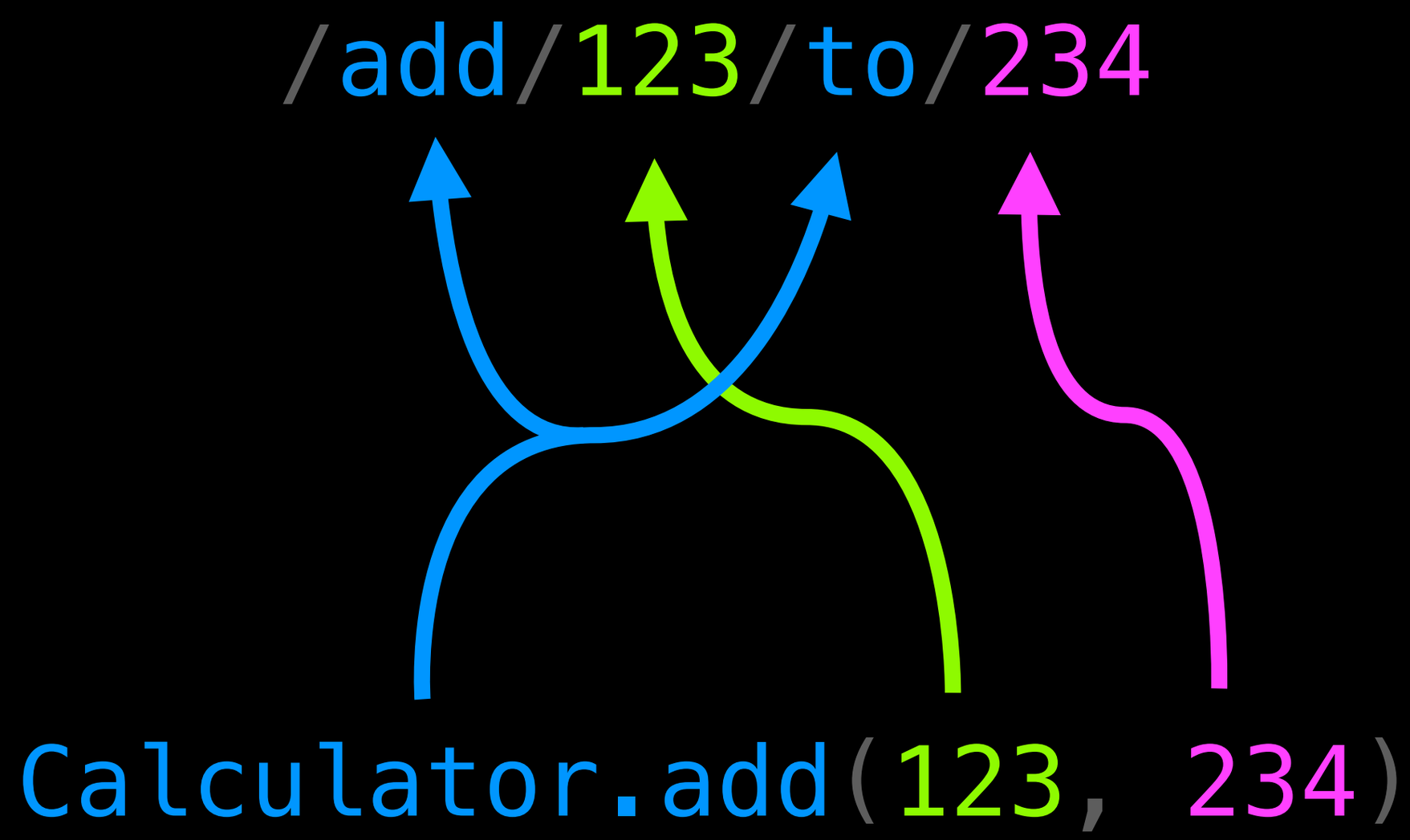
```
Calculator.add(123, 234)
```

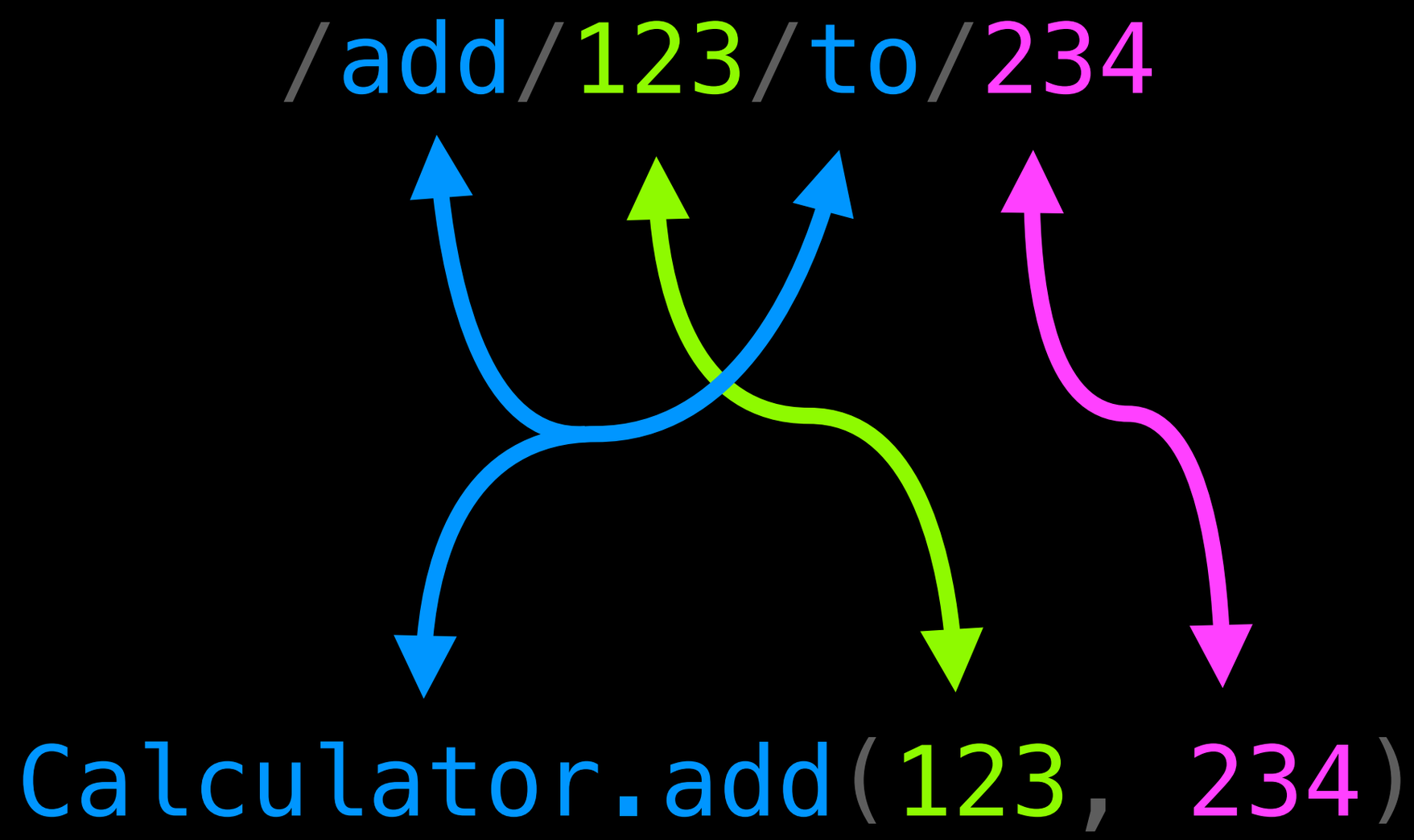
/add/123/to/234



```
Calculator.add(123, 234)
```


/add/123/to/234
Calculator.add(123, 234)


/add/123/to/234
Calculator.add(123, 234)


/add/123/to/234
Calculator.add(123, 234)

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}



Calculator.add.url(1, 2)      // => “/add/1/to/2”
```

talk plan

talk plan

1. path segments

talk plan

1. path segments
2. whole-path representations

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations
4. binding paths to code

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations
4. binding paths to code
5. making it pretty!

talk plan

1. **path segments**
2. whole-path representations
3. whole-path transformations
4. binding paths to code
5. making it pretty!

?

Q: How can we transform between
a segment of a URL path and
any data type we may need in our web app?

/add/123/to/234

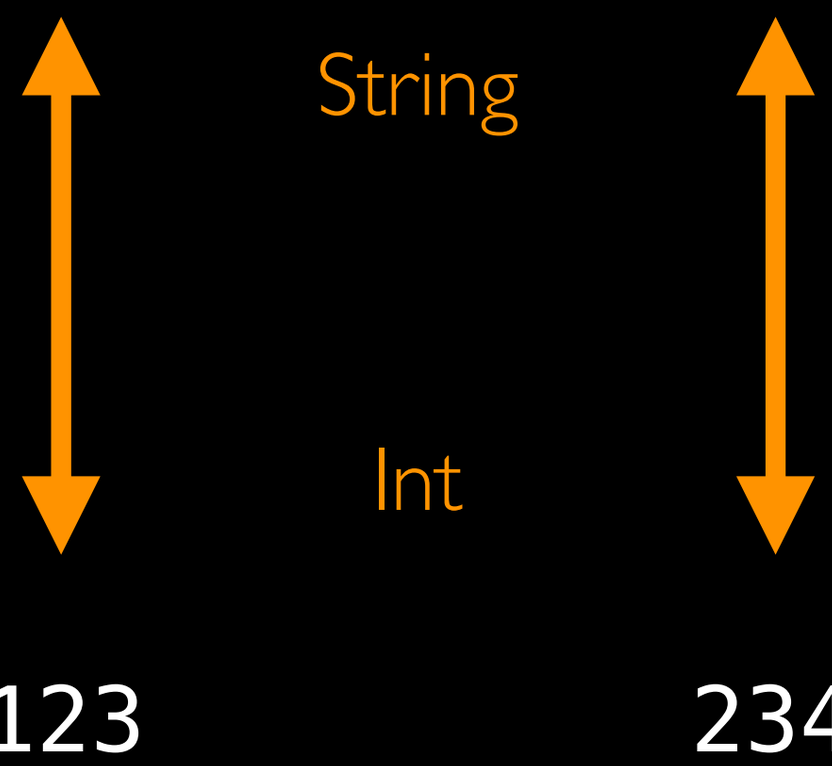
/add/123/to/234

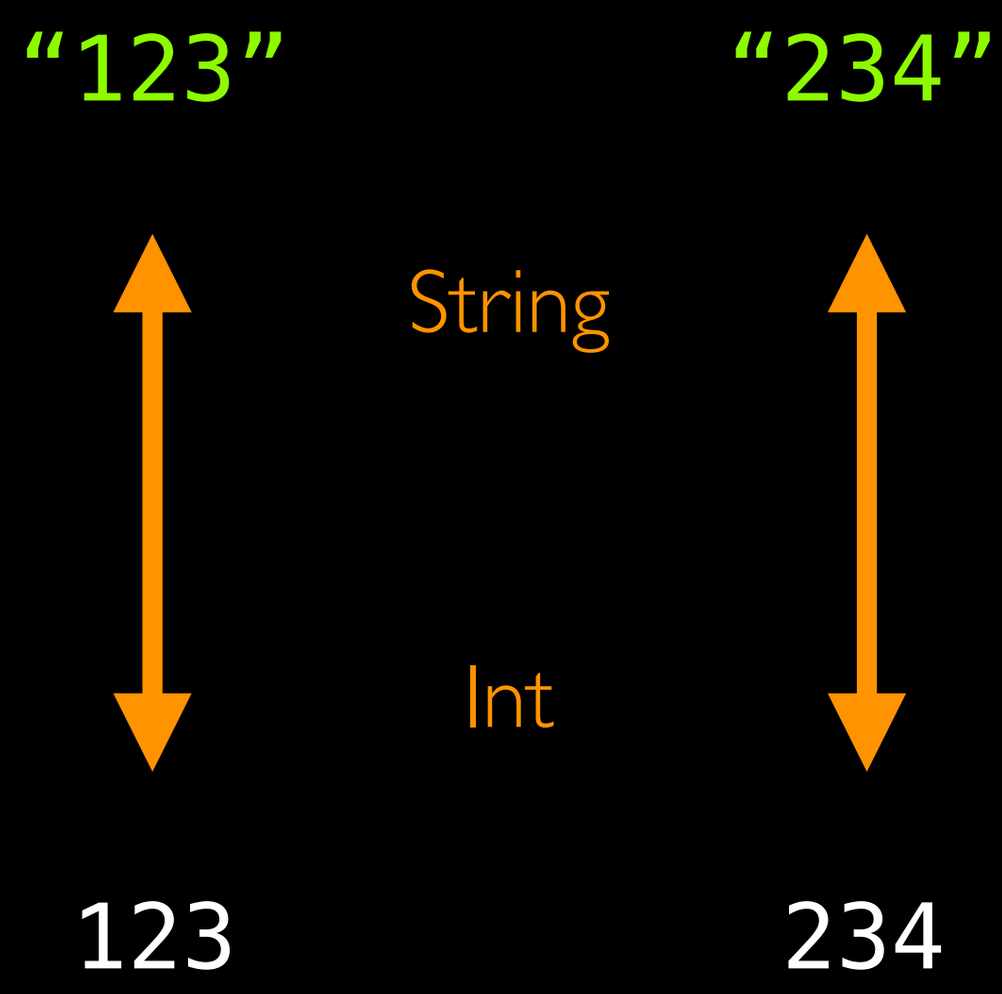
```
List("add", "123", "to", "234")
```

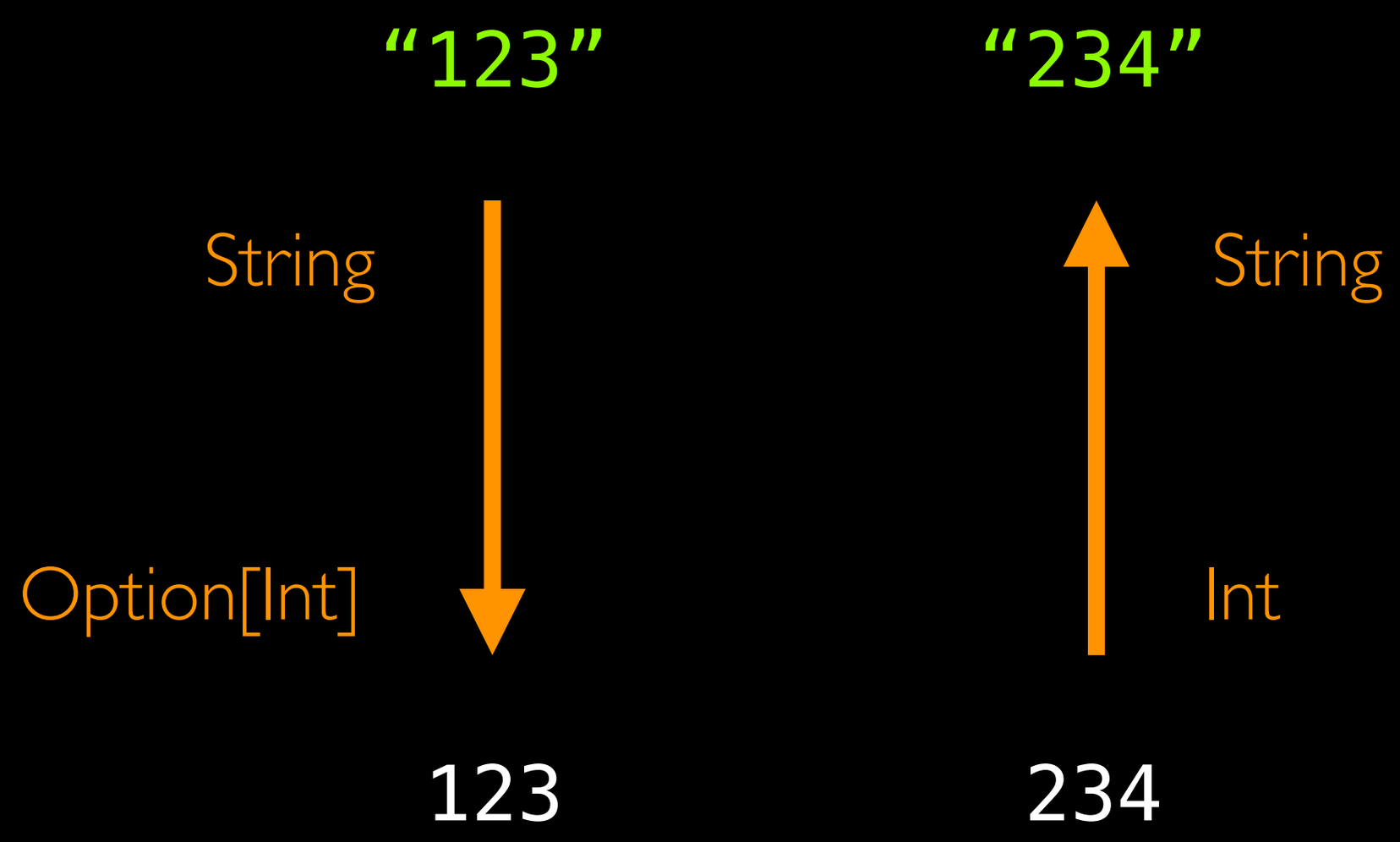
```
"add" :: "123" :: "to" :: "234" :: Nil
```

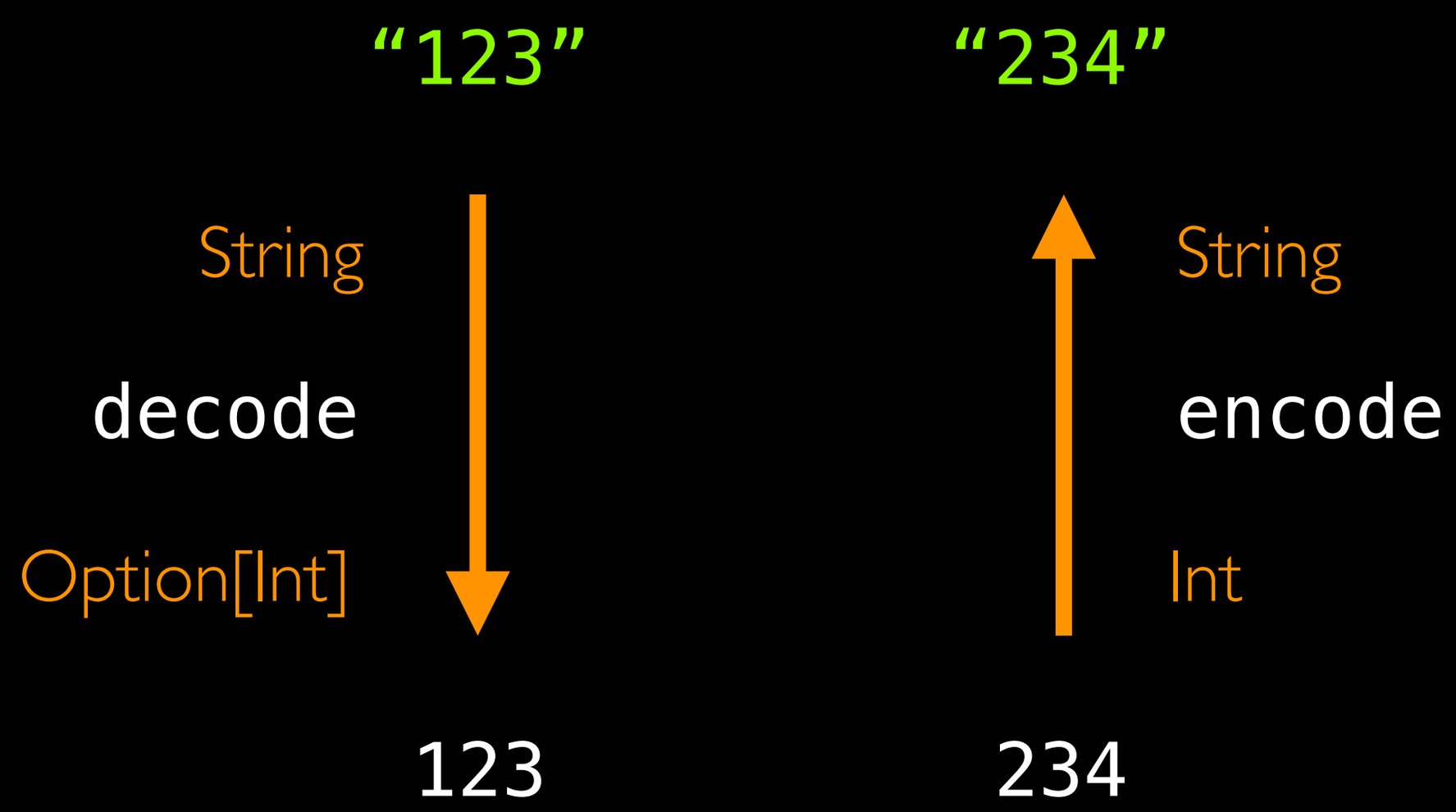
```
List(“add”, “123”, “to”, “234”)
```

```
List("add", "123", "to", "234")
```




"123"
"234"
String
Int
123
234


"123"
"234"
String
String
Option[Int]
Int
123
234


"123"
"234"
String
decode
Option[Int]
String
encode
Int
123
234

```scala
trait Arg[T] {

  /**
   * Attempt to decode a URL path segment.
   *
   * Return Some(value) if successful,
   * or None if unsuccessful.
   */
  def decode(in: String): Option[T]

  /** Encode a typed value as a URL path segment. */
  def encode(in: T): String

}
```

```scala
object IntArg extends Arg[Int] {

  /**
   * Attempt to decode a URL path segment.
   *
   * Return Some(value) if successful,
   * or None if unsuccessful.
   */
  def decode(in: String): Option[Int] =
    try {
      Some(in.toInt)
    } catch {
      case exn: NumberFormatException => None
    }

  /** Encode a typed value as a URL path segment. */
  def encode(in: Int): String =
    in.toString

}
```

```scala
object IntArg extends Arg[Int] {

  /**
   * Attempt to decode a URL path segment.
   *
   * Return Some(value) if successful,
   * or None if unsuccessful.
   */
  def decode(in: String): Option[Int] =
    try {
      Some(in.toInt)
    } catch {
      case exn: NumberFormatException => None
    }

  /** Encode a typed value as a URL path segment. */
  def encode(in: Int): String =
    in.toString

}
```

```scala
object IntArg extends Arg[Int]
```

```scala
object IntArg extends Arg[Int]

object StringArg extends Arg[String]

object DoubleArg extends Arg[Double]

object FooArg extends Arg[Foo]

object FooListArg extends Arg[List[Foo]]

// ...
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}


Calculator.add.url(1, 2)      // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)   // => Response(“abc * 2 = abcabc”)
```

Q: How can we transform between
a segment of a URL path and
any data type we may need in our web app?

Q: How can we transform between
a segment of a URL path and
any data type we may need in our web app?

A: By defining an **Arg** for each type we need

talk plan

1. **path segments**
2. whole-path representations
3. whole-path transformations
4. binding paths to code
5. making it pretty!

talk plan

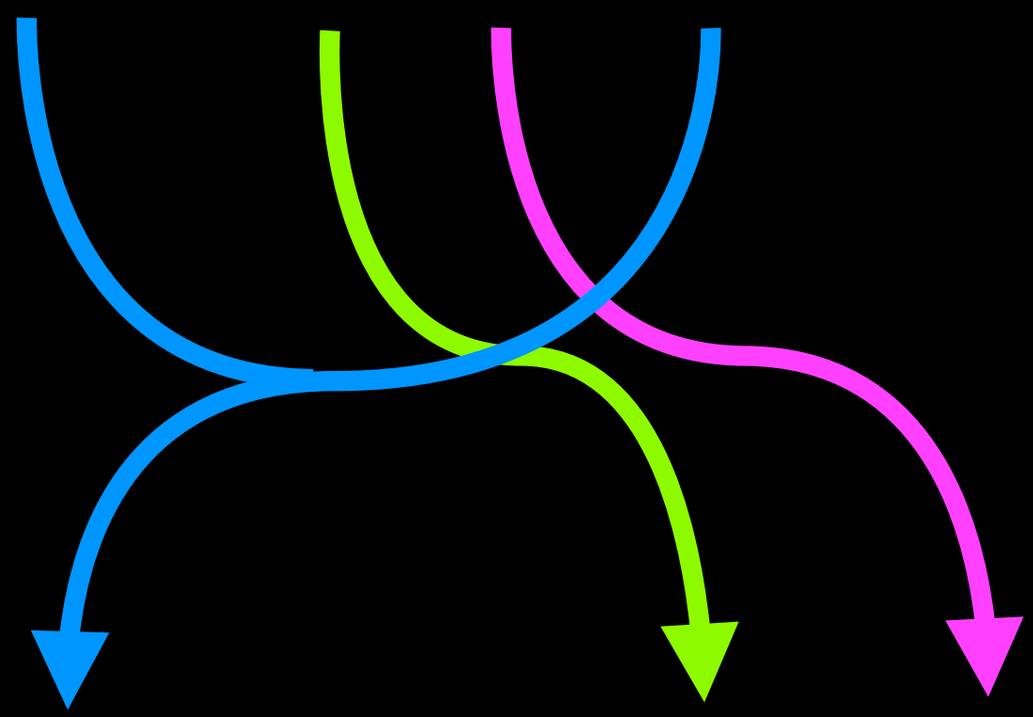
1. path segments
2. **whole-path representations**
3. whole-path transformations
4. binding paths to code
5. making it pretty!

?

Q: How can we aggregate the transformed URL path segments, while maintaining type information?

/repeat/abc/2/times

/repeat/abc/2/times



Calculator.repeat(“abc”, 2)

/repeat/abc/2/times

```
List(“repeat”, “abc”, “2”, “times”)
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

“repeat” :: “abc” :: “2” :: “times” :: Nil

StringArg

IntArg

“abc”

2

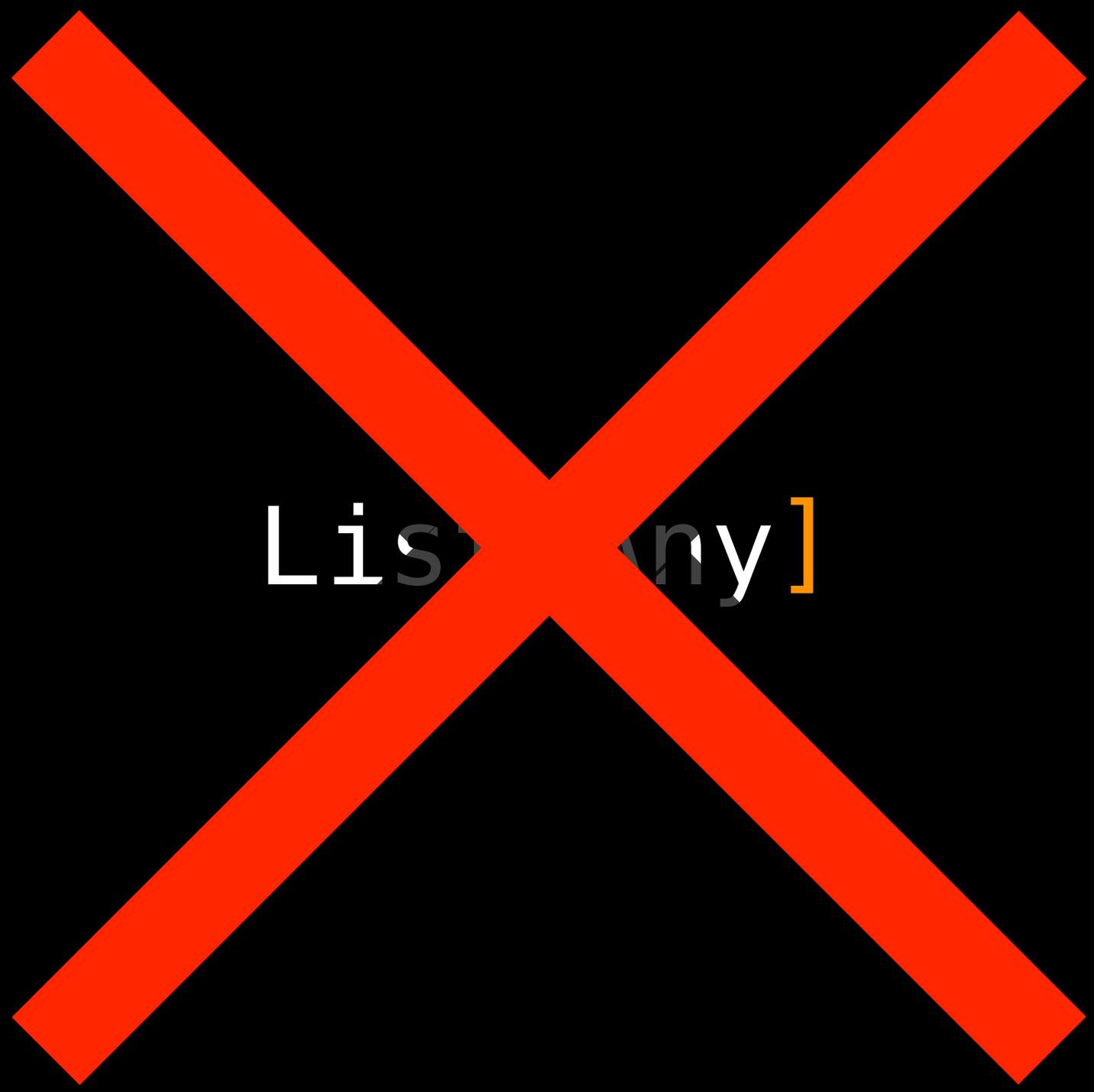
“repeat” :: “abc” :: “2” :: “times” :: Nil

StringArg

IntArg

“abc” :: 2 :: Nil

```
List[Any]
```


List[Any]

“repeat” :: “abc” :: “2” :: “times” :: Nil

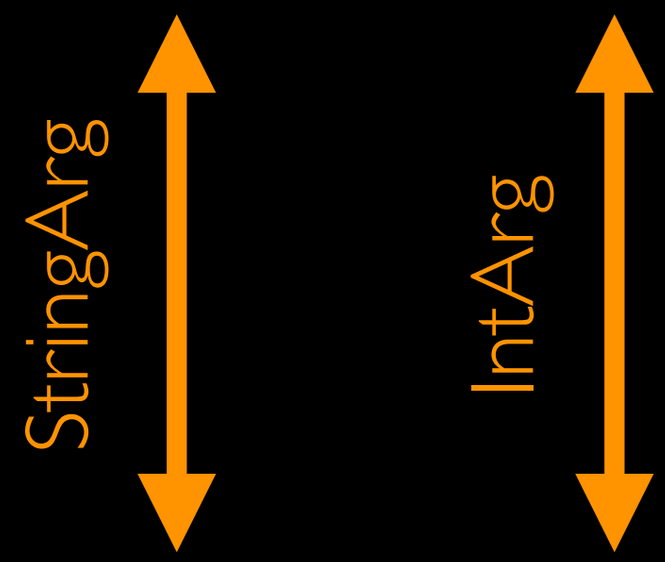
StringArg

IntArg

“abc”

2

“repeat” :: “abc” :: “2” :: “times” :: Nil



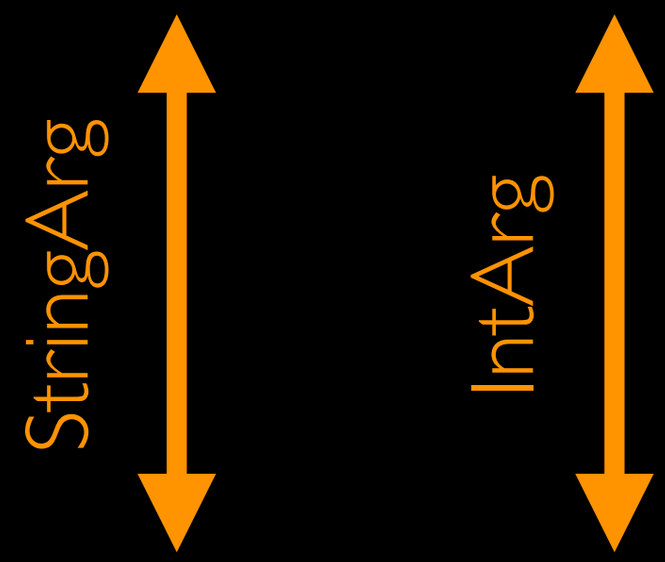
(“abc” , 2)

```
Tuple2[String, Int]
```

```
Tuple2[String, Int]
```

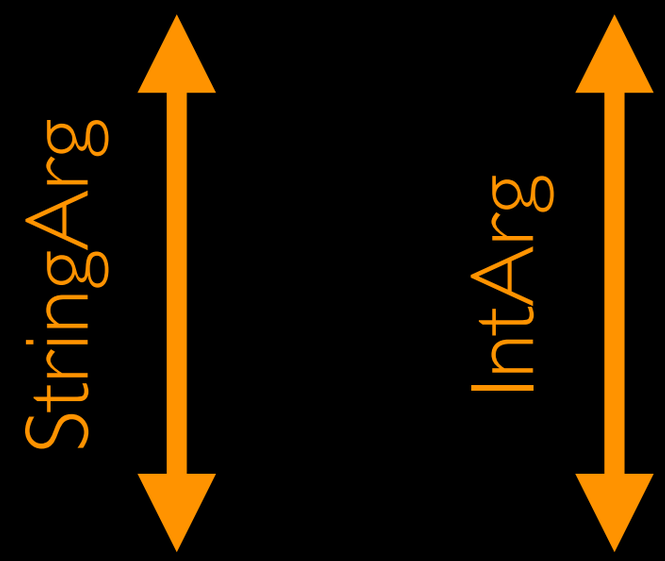
```
Tuple2[String, Int]
```

“repeat” :: “abc” :: “2” :: “times” :: Nil



(“abc” , 2)

“repeat” :: “abc” :: “2” :: “times” :: Nil



(“abc” , (2 , ???))

```
Tuple2[String,
       Tuple2[Int, ???]]
```

```
HCons[String,
      HCons[Int, HNil]]
```

```
HCons[String,
  HCons[Int, HNil]]
```

```scala
trait List[+A] {

  def ::(item: A): List[A] =
    Cons(item, this)

}

class Cons[A](hd: A, tl: List[A])
  extends List[A]

object Nil extends List[Nothing]
```

```scala
trait List[+A] {

  def ::(item: A): List[A] =
    Cons(item, this)

}

class Cons[A](hd: A, tl: List[A])
  extends List[A]

object Nil extends List[Nothing]
```

```scala
trait List[+A] {

  def ::(item: A): List[A] =
    Cons(item, this)

}

class Cons[A](hd: A, tl: List[A])
  extends List[A]

object Nil extends List[Nothing]
```

```scala
trait List[+A] {

  def ::(item: A): List[A] =
    Cons(item, this)

}

class Cons[A](hd: A, tl: List[A])
  extends List[A]

object Nil extends List[Nothing]
```

```scala
trait HList

class HCons[H, T <: HList](hd: H, tl: T)
  extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}

object HNil extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}
```

```scala
trait HList

class HCons[H, T <: HList](hd: H, tl: T)
  extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}

object HNil extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}
```

```scala
trait HList

class HCons[H, T <: HList](hd: H, tl: T)
  extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}

object HNil extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}
```

```scala
trait HList

class HCons[H, T <: HList](hd: H, tl: T)
  extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}

object HNil extends HList {

  def ::[X](item: X) =
    HCons(item, this)

}
```

```
bash$ cd 2-hlists
bash$ ./sbt console
```

...

```
scala>
```

Try it!

```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> “a” :: “b” :: “c” :: Nil
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      List(a, b, c)

scala>
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      Cons(a,Cons(b,Cons(c,Nil)))

scala>
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      Cons(a,Cons(b,Cons(c,Nil)))

scala> "a" :: "b" :: "c" :: HNil
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      Cons(a,Cons(b,Cons(c,Nil)))

scala> "a" :: "b" :: "c" :: HNil
res1: HCons[String,HCons[String,HCons[String,HNil]]] =
      HCons(a,HCons(b,HCons(c,HNil)))

scala>
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      Cons(a,Cons(b,Cons(c,Nil)))

scala> "a" :: "b" :: "c" :: HNil
res1: HCons[String,HCons[String,HCons[String,HNil]]] =
      HCons(a,HCons(b,HCons(c,HNil)))

scala> "a" :: 2 :: 3.0 :: HNil
```


```
bash$ cd 2-hlists
bash$ ./sbt console

                    ...

scala> "a" :: "b" :: "c" :: Nil
res0: List[String] =
      Cons(a,Cons(b,Cons(c,Nil)))

scala> "a" :: "b" :: "c" :: HNil
res1: HCons[String,HCons[String,HCons[String,HNil]]] =
      HCons(a,HCons(b,HCons(c,HNil)))

scala> "a" :: 2 :: 3.0 :: HNil
res2: HCons[String,HCons[Int,HCons[Double,HNil]]] =
      HCons(a,HCons(2,HCons(3.0,HNil)))
```

Q: How can we aggregate the transformed URL path segments, while maintaining type information?

Q: How can we aggregate the transformed URL path segments, while maintaining type information?

A: By defining an **HList** that contains the relevant types

talk plan

1. path segments
2. **whole-path representations**
3. whole-path transformations
4. binding paths to code
5. making it pretty!

talk plan

1. path segments
2. whole-path representations
3. **whole-path transformations**
4. binding paths to code
5. making it pretty!

?

Q: How can we create a bidirectional, type-safe mapping between a URL and a particular type of HList?

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

```
“abc” :: 2 :: HNil
```

“repeat” :: “abc” :: “2” :: “times” :: Nil

“abc” :: 2 :: HNil

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

```
“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil
```

```
“abc” :: 2 :: HNil
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil

“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil

“abc” :: 2 :: HNil
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]
}

class PLiteral[T <: Path](val head: String, val tail: T)
  extends Path {
  type Result = tail.Result
  def :/:(head: String) = PLiteral(head, this)
  def :/:[X](head: Arg[X]) = PMatch(head, this)
}

class PMatch[H, T <: Path](val head: Arg[H], val tail: T)
  extends Path {
  type Result = HCons[H, tail.Result]
  def :/:(...) ...
}

object PNil extends Path {
  type Result = HNil
  def :/:(...) ...
}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]
}

class PLiteral[T <: Path](val head: String, val tail: T)
  extends Path {
  type Result = tail.Result
  def :/:(head: String) = PLiteral(head, this)
  def :/:[X](head: Arg[X]) = PMatch(head, this)
}

class PMatch[H, T <: Path](val head: Arg[H], val tail: T)
  extends Path {
  type Result = HCons[H, tail.Result]
  def :/:(...) ...
}

object PNil extends Path {
  type Result = HNil
  def :/:(...) ...
}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]
}

class PLiteral[T <: Path](val head: String, val tail: T)
  extends Path {
  type Result = tail.Result
  def :/:(head: String) = PLiteral(head, this)
  def :/:[X](head: Arg[X]) = PMatch(head, this)
}

class PMatch[H, T <: Path](val head: Arg[H], val tail: T)
  extends Path {
  type Result = HCons[H, tail.Result]
  def :/:(...) ...
}

object PNil extends Path {
  type Result = HNil
  def :/:(...) ...
}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]
}

class PLiteral[T <: Path](val head: String, val tail: T)
  extends Path {
  type Result = tail.Result
  def :/:(head: String) = PLiteral(head, this)
  def :/:[X](head: Arg[X]) = PMatch(head, this)
}

class PMatch[H, T <: Path](val head: Arg[H], val tail: T)
  extends Path {
  type Result = HCons[H, tail.Result]
  def :/:(...) ...
}

object PNil extends Path {
  type Result = HNil
  def :/:(...) ...
}
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

```
“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil
```

```
“abc” :: 2 :: HNil
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

```
“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil
```

```
“abc” :: 2 :: HNil
```

“repeat” :: “abc” :: “2” :: “times” :: **Nil**

“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil

“abc” :: 2 :: **HNil**

```
type Result = HNil
```

“repeat” :: “abc” :: “2” :: “times” :: **Nil**

“repeat” :/: StringArg :/: IntArg :/: “times” :/: **PNil**

“abc” :: 2 :: **HNil**

```
type Result = HNil
decode: (List[String]) => Option[HNil]
encode: HNil => List[String]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

String

T

PLiteral[T <: Path]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PNil
"times"	PNil

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PLiteral[PNil]

```
"abc" :: 2 :: HNil
```

“repeat” :: “abc” :: “2” :: **“times” :: Nil**

String PNil

“repeat” :/: StringArg :/: IntArg :/: **“times” :/: PNil**

PLiteral[PNil]

“abc” :: 2 :: **HNil**

```
type Result = tail.Result
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PNil
"times"	PNil

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PLiteral[PNil]

```
"abc" :: 2 :: HNil
```

```
type Result = HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

String

PNil

PLiteral[PNil]

```
"abc" :: 2 :: HNil
```

```
type Result = HNil
decode: (List[String]) => Option[HNil]
encode: HNil => List[String]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil

"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil

"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

```
                          Arg[H]          T
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[H,T <: Path]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[Int] PLiteral[PNil]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[Int, PLiteral[PNil]]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[Int] PLiteral[PNil]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[Int, PLiteral[PNil]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[H, tail.Result]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[Int] PLiteral[PNil]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[Int, PLiteral[PNil]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[Int, HNil]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[Int] PLiteral[PNil]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[Int, PLiteral[PNil]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[Int, HNil]
decode: (List[String]) => Option[HCons[Int, HNil]]
encode: HCons[Int, HNil] => List[String]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil

"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil

"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

	Arg[H]	T	
"repeat" :/:	StringArg :/:	IntArg :/:	"times" :/: PNil

PMatch[H,T <: Path]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[String] PMatch[Int, PLiteral[PNil]]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[String, PMatch[Int, PLiteral[PNil]]]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[String] PMatch[Int, PLiteral[PNil]]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[String, PMatch[Int, PLiteral[PNil]]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[H, tail.Result]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[String] PMatch[Int, PLiteral[PNil]]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[String, PMatch[Int, PLiteral[PNil]]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[String, HCons[Int, HNil]]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

Arg[String] PMatch[Int, PLiteral[PNil]]

```
"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil
```

PMatch[String, PMatch[Int, PLiteral[PNil]]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[String, HCons[Int, HNil]]
decode: (List[String]) => Option[HCons[String, HCons[Int, HNil]]]
encode: HCons[String, HCons[Int, HNil]] => List[String]
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil

“repeat” :/: StringArg :/: IntArg :/: “times” :/: PNil

“abc” :: 2 :: HNil
```

```
“repeat” :: “abc” :: “2” :: “times” :: Nil
```

String	T
“repeat” :/:	StringArg :/: IntArg :/: “times” :/: PNil

PLiteral[T <: Path]

```
“abc” :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PMatch[String, PMatch[Int, PLiteral[PNil]]]
`"repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil`	

PLiteral[PMatch[String, PMatch[Int, PLiteral[PNil]]]]

```
"abc" :: 2 :: HNil
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PMatch[String, PMatch[Int, PLiteral[PNil]]]
`"repeat" :/:`	`StringArg :/: IntArg :/: "times" :/: PNil`

PLiteral[PMatch[String, PMatch[Int, PLiteral[PNil]]]]

```
"abc" :: 2 :: HNil
```

```
type Result = tail.Result
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PMatch[String, PMatch[Int, PLiteral[PNil]]]
`"repeat" :/:`	`StringArg :/: IntArg :/: "times" :/: PNil`

PLiteral[PMatch[String, PMatch[Int, PLiteral[PNil]]]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[String, HCons[Int, HNil]]
```

```
"repeat" :: "abc" :: "2" :: "times" :: Nil
```

String	PMatch[String, PMatch[Int, PLiteral[PNil]]]
"repeat" :/:	StringArg :/: IntArg :/: "times" :/: PNil

PLiteral[PMatch[String, PMatch[Int, PLiteral[PNil]]]]

```
"abc" :: 2 :: HNil
```

```
type Result = HCons[String, HCons[Int, HNil]]
decode: (List[String]) => Option[HCons[String, HCons[Int, HNil]]]
encode: HCons[String, HCons[Int, HNil]] => List[String]
```

```
bash$ cd 3-paths
bash$ ./sbt console

...

scala>
```

Try it!

Try it!

```
bash$ cd 3-paths
bash$ ./sbt console
```

...

```
scala> val p = “repeat” :/: StringArg :/:
               IntArg :/: “times”   :/: PNil
```


```
bash$ cd 3-paths
bash$ ./sbt console

...

scala> val p = “repeat” :/: StringArg :/:
                IntArg :/: “times”   :/: PNil
p: PLiteral[PMatch[String,PMatch[Int,PLiteral[PNil]]]] =
   PLiteral(repeat,PMatch(StringArg,
            PMatch(IntArg, PLiteral(times,PNil))))

scala>
```


```
bash$ cd 3-paths
bash$ ./sbt console
```

...

```
scala> val p = “repeat” :/: StringArg :/:
                 IntArg :/: “times”   :/: PNil
p: PLiteral[PMatch[String,PMatch[Int,PLiteral[PNil]]]] =
   PLiteral(repeat,PMatch(StringArg,
            PMatch(IntArg, PLiteral(times,PNil))))

scala> p.decode(“repeat” :: “abc” :: “2” :: “times” :: Nil)
```


```
bash$ cd 3-paths
bash$ ./sbt console

...

scala> val p = “repeat” :/: StringArg :/:
               IntArg :/: “times”   :/: PNil
p: PLiteral[PMatch[String,PMatch[Int,PLiteral[PNil]]]] =
   PLiteral(repeat,PMatch(StringArg,
            PMatch(IntArg, PLiteral(times,PNil))))

scala> p.decode(“repeat” :: “abc” :: “2” :: “times” :: Nil)
res0: Option[p.Result] =
      Some(HCons(abc,HCons(2,HNil)))

scala>
```

Try it!

```
bash$ cd 3-paths
bash$ ./sbt console
```

...

```
scala> val p = “repeat” :/: StringArg :/:
                IntArg :/: “times”   :/: PNil
p: PLiteral[PMatch[String,PMatch[Int,PLiteral[PNil]]]] =
   PLiteral(repeat,PMatch(StringArg,
            PMatch(IntArg, PLiteral(times,PNil))))

scala> p.decode(“repeat” :: “abc” :: “2” :: “times” :: Nil)
res0: Option[p.Result] =
      Some(HCons(abc,HCons(2,HNil)))

scala> p.encode(“abc” :: 2 :: HNil)
```

Try it!

```
bash$ cd 3-paths
bash$ ./sbt console
```

...

```
scala> val p = “repeat” :/: StringArg :/:
                 IntArg :/: “times”   :/: PNil
p: PLiteral[PMatch[String,PMatch[Int,PLiteral[PNil]]]] =
   PLiteral(repeat,PMatch(StringArg,
            PMatch(IntArg, PLiteral(times,PNil))))

scala> p.decode(“repeat” :: “abc” :: “2” :: “times” :: Nil)
res0: Option[p.Result] =
      Some(HCons(abc,HCons(2,HNil)))

scala> p.encode(“abc” :: 2 :: HNil)
res1: List[String] = List(repeat, abc, 2, times)
```

```
bash$ cd 3-paths
bash$ ./sbt console

...

scala>
```

Try it!

Try it!

```
bash$ cd 3-paths
bash$ ./sbt console

...

scala> p.encode("abc" :: "2" :: HNil)
```


```
bash$ cd 3-paths
bash$ ./sbt console

                    ...

scala> p.encode(“abc” :: “2” :: HNil)
<console>:9: error: type mismatch;
 found   : HCons[String,HCons[String,HNil]]
 required: p.Result
            p.encode("abc" :: "2" :: HNil)
                           ^
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}


Calculator.add.url(1, 2)      // => “/add/1/to/2”

Calculator.multiply(3, 4)     // => Response(“3 * 4 = 12”)
```

Q: How can we create a bidirectional, type-safe mapping between a URL and a particular type of HList?

Q: How can we create a bidirectional, type-safe mapping between a URL and a particular type of HList?

A: By defining a **Path** with the correct **Result** type

talk plan

1. path segments
2. whole-path representations
3. **whole-path transformations**
4. binding paths to code
5. making it pretty!

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations
4. **binding paths to code**
5. making it pretty!

part 1 - routes

?

Q: How can we call a function and pass it the typed data we have extracted from a URL?

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]



}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response): Route[Result] =
    new Route(this, fn)

}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response): Route[Result] =
    new Route(this, fn)

}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response): Route[Result] =
    new Route(this, fn)

}
```

```scala
class Route[Arg <: HList](
  val path: Path { type Result = Arg },
  val fn: (Arg) => Response
) {

  def dispatch(req: Request): Option[Response] =
    path.decode(req.path).map(fn)

  def url(arg: Arg): String =
    path.encode(arg).
         map(urlEncode(_, “utf-8”)).
         mkString("/", "/", "")

}
```

```scala
class Route[Arg <: HList](
  val path: Path { type Result = Arg },
  val fn: (Arg) => Response
) {

  def dispatch(req: Request): Option[Response] =
    path.decode(req.path).map(fn)

  def url(arg: Arg): String =
    path.encode(arg).
         map(urlEncode(_, “utf-8”)).
         mkString("/", "/", "")

}
```

```scala
class Route[Arg <: HList](
  val path: Path { type Result = Arg },
  val fn: (Arg) => Response
) {

  def dispatch(req: Request): Option[Response] =
    path.decode(req.path).map(fn)

  def url(arg: Arg): String =
    path.encode(arg).
         map(urlEncode(_, “utf-8”)).
         mkString("/", "/", "")

}
```

```scala
class Route[Arg <: HList](
  val path: Path { type Result = Arg },
  val fn: (Arg) => Response
) {

  def dispatch(req: Request): Option[Response] =
    path.decode(req.path).map(fn)

  def url(arg: Arg): String =
    path.encode(arg).
         map(urlEncode(_, “utf-8”)).
         mkString("/", "/", "")

}
```

```scala
class Route[Arg <: HList](
  val path: Path { type Result = Arg },
  val fn: (Arg) => Response
) {

  def dispatch(req: Request): Option[Response] =
    path.decode(req.path).map(fn)

  def url(arg: Arg): String =
    path.encode(arg).
         map(URLEncoder.encode(_, “utf-8”)).
         mkString("/", "/", "")

}
```

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala>
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val path = "add" :/: IntArg :/:
                  "to" :/: IntArg :/: PNil
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala> val path = "add" :/: IntArg :/:
                  "to" :/: IntArg :/: PNil
path: PLiteral[PArg[Int,PLiteral[PArg[Int,PNil]]]] =
      PLiteral(add,PArg(IntArg,PLiteral(to,
        PArg(IntArg,PNil))))

scala>
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val path = "add" :/: IntArg :/:
                  "to" :/: IntArg :/: PNil
path: PLiteral[PArg[Int,PLiteral[PArg[Int,PNil]]]] =
      PLiteral(add,PArg(IntArg,PLiteral(to,
        PArg(IntArg,PNil))))

scala> def fn(res: HCons[Int, HCons[Int, HNil]]) = {
         val a = res.head
         val b = res.tail.head
         val ans = a + b
         Response("%s + %s = %s".format(a, b, a + b)
       }
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val path = "add" :/: IntArg :/:
                  "to" :/: IntArg :/: PNil
path: PLiteral[PArg[Int,PLiteral[PArg[Int,PNil]]]] =
      PLiteral(add,PArg(IntArg,PLiteral(to,
        PArg(IntArg,PNil))))

scala> def fn(res: HCons[Int, HCons[Int, HNil]]) = {
         val a = res.head
         val b = res.tail.head
         val ans = a + b
         Response("%s + %s = %s".format(a, b, a + b)
       }
fn: (res: HCons[Int,HCons[Int,HNil]])Response
```

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala>
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala> val route = path >> fn
```


```
bash$ cd 4-routes
bash$ ./sbt console

                    ...

scala> val route = path >> fn
route: Route[path.Result] =
        Route(PLiteral(add,PArg(IntArg,PLiteral(to,
          PArg(IntArg,PNil)))),<function1>)

scala>
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val route = path >> fn
route: Route[path.Result] =
        Route(PLiteral(add,PArg(IntArg,PLiteral(to,
          PArg(IntArg,PNil)))),<function1>)

scala> route.dispatch(Request("/add/123/to/234"))
```


```
bash$ cd 4-routes
bash$ ./sbt console

                    ...

scala> val route = path >> fn
route: Route[path.Result] =
        Route(PLiteral(add,PArg(IntArg,PLiteral(to,
          PArg(IntArg,PNil)))),<function1>)

scala> route.dispatch(Request("/add/123/to/234"))
res0: Option[Response] = Some(Response("123 + 234 = 357"))

scala>
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val route = path >> fn
route: Route[path.Result] =
        Route(PLiteral(add,PArg(IntArg,PLiteral(to,
          PArg(IntArg,PNil)))),<function1>)

scala> route.dispatch(Request("/add/123/to/234"))
res0: Option[Response] = Some(Response("123 + 234 = 357"))

scala> route.dispatch(Request("/add/abc/to/234"))
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> val route = path >> fn
route: Route[path.Result] =
        Route(PLiteral(add,PArg(IntArg,PLiteral(to,
          PArg(IntArg,PNil)))),<function1>)

scala> route.dispatch(Request("/add/123/to/234"))
res0: Option[Response] = Some(Response("123 + 234 = 357"))

scala> route.dispatch(Request("/add/abc/to/234"))
res1: Option[Response] = None
```

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala>
```

Try it!

```
bash$ cd 4-routes
bash$ ./sbt console

...

scala> route.url(123 :: 234 :: HNil)
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> route.url(123 :: 234 :: HNil)
res2: String = "/add/123/to/234"

scala>
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> route.url(123 :: 234 :: HNil)
res2: String = "/add/123/to/234"

scala> route.url("123" :: 234 :: HNil)
```


```
bash$ cd 4-routes
bash$ ./sbt console
```

...

```
scala> route.url(123 :: 234 :: HNil)
res2: String = "/add/123/to/234"

scala> route.url("123" :: 234 :: HNil)
<console>:13: error: type mismatch;
 found   : String("123")
 required: Int
       route.url("123" :: 234 :: HNil)
                 ^
```

Q: How can we call a function and pass it the typed data we have extracted from a URL?

Q: How can we call a function and pass it the typed data we have extracted from a URL?

A: By constructing a **Route** that combines a Path and a Function of corresponding types

part 2 - sites

?

Q: How can we use the contents of a URL to choose which Route we should take?

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to"  /: IntArg :/: PNil) >> {
    ("add" :/: IntArg :/: "to"  /: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons  t, HNil]]) =>
        val a = args.head
        val b = args.tail.hea
        Response("%s + %s = %  format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/  IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HC  [Int, HNil]]) =>
        val a = args.head
        val b = args.tail.hea
        Response("%s * %s =  ormat(a, b, a * b))
    }

}
```

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
trait Path {

  type Result <: HList

  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response): Route[Result] = {
    new Route(this, fn)
  }

}
```

```scala
trait Path {

  type Result <: HList

  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response)                    :
        Route[Result] = {
                  new Route(this, fn)

  }

}
```

```scala
trait Path {
  type Result <: HList
  def decode(path: List[String]): Option[Result]
  def encode(args: Result): List[String]

  def >>(fn: (Result) => Response)(implicit site: Site):
        Route[Result] = {
    site.addRoute(new Route(this, fn))
    route
  }

}
```

```scala
trait Site {

  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...

}
```

```scala
trait Site {
  implicit val site: Site = this
  var routes: List[Route[_]] = Nil

  def addRoute(route: Route[_]): Unit =
    routes = routes :+ route

  def dispatch(req: Request): Option[Response] =
    ...
}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }(site)

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }(site)

}
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

Q: How can we use the contents of a URL to choose which Route we should take?

Q: How can we use the contents of a URL to choose which Route we should take?

A: By registering several Route with a **Site** (and using implicits to paper over the cracks)

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations
4. **binding paths to code**
5. making it pretty!

talk plan

1. path segments
2. whole-path representations
3. whole-path transformations
4. binding paths to code
5. **making it pretty!**

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}



Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}



Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: PNil) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: PNil) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”
```

?

Q: How can we save the application programmer from dealing with HLists?

Q: How can we save the application programmer from dealing with HLists?

Q: How can we save the application programmer from dealing with HLists?

A: Through judicious use of pimping and implicits... (check the example *6-implicits* for details)

```scala
class RouteN[A, B, ...](
  path: Path { ... },
  fn: (a: A, b: B, ...) => Response
) extends Route(...) {

  def url(a: A, b: B, ...) =
    path.encode(a :: b :: HNil).mkString("/", "/", "")

  def apply(a: A, b: B, ...) =
    fn.apply(arg)

}
```

```scala
class RouteN[A, B, ...](
  path: Path { ... },
  fn: (a: A, b: B, ...) => Response
) extends Route(...) {

  def url(a: A, b: B, ...) =
    path.encode(a :: b :: HNil).mkString("/", "/", "")

  def apply(a: A, b: B, ...) =
    fn.apply(arg)

}

case class Route0 // ...

case class Route1[A] // ...

case class Route2[A, B] // ...

case class Route3[A, B, C] // ...

case class Route4[A, B, C, D] // ...

case class Route5[A, B, C, D, E] // ...

case class Route6[A, B, C, D, E, F] // ...
```

```scala
trait PimpedPath {

  type Rt <: Route

  def >>[A, B, ...](fn: (A, B, ...) => Response)
    (implicit site: Site): Rt

}
```

```scala
trait PimpedPath {

  type Rt <: Route

  def >>[A, B, ...](fn: (A, B, ...) => Response)
    (implicit site: Site): Rt

}

implicit def pimpPath0](...) // ...

implicit def pimpPath1[A](...) // ...

implicit def pimpPath2[A, B](...) // ...

implicit def pimpPath3[A, B, C](...) // ...

implicit def pimpPath4[A, B, C, D](...) // ...

implicit def pimpPath5[A, B, C, D, E](...) // ...

implicit def pimpPath6[A, B, C, D, E, F](...) // ...

implicit def pimpPath7[A, B, C, D, E, F, G](...) // ...

implicit def pimpPath8[A, B, C, D, E, F, G, H](...) // ...
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”
```

```
bash$ cd 6-implicits
bash$ ./sbt test
```

```
bash$ cd 7-lift-app
bash$ ./sbt jetty-run ~prepare-webapp
```

Try it!

```
bash$ cd 7-lift-app
bash$ ./sbt jetty-run ~prepare-webapp
```

Try it!

extensions

extensions

1. rest arguments `/append/a/b/c/...`

extensions

1. rest arguments `/append/a/b/c/...`
2. request parameters `?a=b`

extensions

1. rest arguments `/append/a/b/c/...`
2. request parameters `?a=b`
3. generating full links to routes

```
<a href="/foo" title="">...</a>
```

extensions

1. rest arguments `/append/a/b/c/...`
2. request parameters `?a=b`
3. generating full links to routes

```
<a href="/foo" title="">...</a>
```

4. route-level access control

references

references

Original paper on HLists in Haskell:
Oleg Kiselyov, Ralf Lämmel, and Keean Schupke
Strongly Typed Heterogenous Collections
http://homepages.cwi.nl/~ralf/HList/

references

Original paper on HLists in Haskell:
Oleg Kiselyov, Ralf Lämmel, and Keean Schupke
Strongly Typed Heterogenous Collections
http://homepages.cwi.nl/~ralf/HList/

Awesome blog posts on Type-Level Programming and HLists in Scala:
Mark Harrah
Type-level programming in Scala
http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala

references

The delectable term "pimping" (applied to Scala):
Martin Odersky
Pimp my Library
http://www.artima.com/weblogs/viewpost.jsp?thread=179766

references

The delectable term "pimping" (applied to Scala):
Martin Odersky
Pimp my Library
http://www.artima.com/weblogs/viewpost.jsp?thread=179766

Everything we've covered here (and more) but written in Racket:
Dave Gurnell
Dispatch.PLT: Binding URLs to Procedures
http://planet.racket-lang.org/package-source/untyped/dispatch.plt/2/1/ ...
... planet-docs/dispatch/index.html

get in touch

Slides, nodes, and code samples

https://github.com/davegurnell/scalalol-2011-talk

get in touch

Slides, nodes, and code samples
https://github.com/davegurnell/scalalol-2011-talk

Bigtop web toolkit
http://bigtopweb.com/routes

get in touch

Slides, nodes, and code samples
https://github.com/davegurnell/scalalol-2011-talk

Bigtop web toolkit
http://bigtopweb.com/routes

Dave Gurnell
dave@untyped.com
@davegurnell

get in touch

Slides, nodes, and code samples
https://github.com/davegurnell/scalalol-2011-talk

Bigtop web toolkit
http://bigtopweb.com/routes

Dave Gurnell
dave@untyped.com
@davegurnell

Untyped
http://untyped.com
@untyped

thanks for listening!

Slides, nodes, and code samples
https://github.com/davegurnell/scalalol-2011-talk

Bigtop web toolkit
http://bigtopweb.com/routes

Dave Gurnell
dave@untyped.com
@davegurnell

Untyped
http://untyped.com
@untyped

unused slides

from section 5

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
def fn(args: HCons[Int, HCons[Int, HNil]]) = {
  val a = args.head
  val b = args.tail.head
  Response("%s + %s = %s".format(a, b, a + b))
}
```

```scala
def fn(args: HCons[Int, HCons[Int, HNil]]) = {
  val a = args.head
  val b = args.tail.head
  Response("%s + %s = %s".format(a, b, a + b))
}
```

```scala
def fn(a: Int, b: Int) =
  Response("%s + %s = %s".format(a, b, a + b))
```

```scala
def fn(args: HCons[Int, HCons[Int, HNil]]) = {
  val a = args.head
  val b = args.tail.head
  Response("%s + %s = %s".format(a, b, a + b))
}
```

```scala
def fn(a: Int, b: Int) =
  Response("%s + %s = %s".format(a, b, a + b))
```

```scala
def convert(in: (Int, Int) => Response):
  (HCons[Int, HCons[Int, HNil]]) => Response
```

```scala
def fn(args: HCons[Int, HCons[Int, HNil]]) = {
  val a = args.head
  val b = args.tail.head
  Response("%s + %s = %s".format(a, b, a + b))
}
```

```scala
def fn(a: Int, b: Int) =
  Response("%s + %s = %s".format(a, b, a + b))
```

```scala
def convert[A, B, R](in: (A, B) => R):
    (HCons[A, HCons[B, HNil]]) => R
```

```scala
def hlistFunction2[A, B, Res](fn: (A, B) => Res) =
  (in: HCons[A, HCons[B, HNil]]) => {
    val h1 = in.head
    val t1 = in.tail
    val h2 = t1.head

    fn(h1, h2)
  }
```

```scala
def hlistFunction2[A, B, Res](fn: (A, B) => Res) =
  (in: HCons[A, HCons[B, HNil]]) => // ...
```

```scala
def hlistFunction0[Res](fn: () => Res) =
  (in: HNil) => // ...

def hlistFunction1[A, Res](fn: (A) => Res) =
  (in: HCons[A, HNil]) => // ...

def hlistFunction2[A, B, Res](fn: (A, B) => Res) =
  (in: HCons[A, HCons[B, HNil]]) => // ...

def hlistFunction3[A, B, C, Res](fn: (A, B, C) => Res) =
  (in: HCons[A, HCons[B, HCons[C, HNil]]]) => // ...

def hlistFunction4[A, B, C, D, Res](fn: (A, B, C, D) => Res) =
  (in: HCons[A, HCons[B, HCons[C, HCons[D, HNil]]]]) => // ...

def hlistFunction5[A, B, C, D, E, Res](fn: (A, B, C, D, E) => Res) =
  (in: HCons[A, HCons[B, HCons[C, HCons[D, HCons[E, HNil]]]]]) => //

def hlistFunction6[A, B, C, D, E, F, Res](fn: (A, B, C, D, E, F) =>
  (in: HCons[A, HCons[B, HCons[C, HCons[D, HCons[E, HCons[F, HNil]]]
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (args: HCons[Int, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (args: HCons[String, HCons[Int, HNil]]) =>
        val a = args.head
        val b = args.tail.head
        Response("%s * %s = %s".format(a, b, a * b))
    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      hlistFunction2((a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b)))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      hlistFunction2((a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b)))


    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      hlistFunction2((a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b)))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      hlistFunction2((a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b)))


    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.url((1, 2))            // => “/add/1/to/2”

Calculator.repeat((“abc”, 2))         // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s = %s".format(a + b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.toURL((1, 2))          // => "/add/1/to/2"

Calculator.repeat(("abc", 2))         // => Response("...")
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
class Route[Res <: HList](
  site: Site,
  path: Path { type Result = Res },
  fn: (Res) => Response
) {
  site.addRoute(this)

  def dispatch(req: Request) =
    path.decode(req.path).map(fn)

  // ...
}
```

```scala
class Route[Res <: HList](
  site: Site,
  path: Path { type Result = Res },
  fn: (Res) => Response
) {
  site.addRoute(this)

  def dispatch(req: Request) =
    path.decode(req.path).map(fn)

  // ...
}

case class Route2[A, B](
  val site: Site,
  val path: Path { type Result = Res },
  val fn: (A, B) => Response
) extends Route(site, path, hlistFunction2(fn)) {

  def url(a: A, b: B) =
    path.encode(a :: b :: HNil).mkString("/", "/", "")

  def apply(a: A, b: B) =
    fn.apply(arg)
}
```

```scala
case class Route2[A, B] // ...
```

```scala
case class Route0 // ...
case class Route1[A] // ...
case class Route2[A, B] // ...
case class Route3[A, B, C] // ...
case class Route4[A, B, C, D] // ...
case class Route5[A, B, C, D, E] // ...
case class Route6[A, B, C, D, E, F] // ...
case class Route7[A, B, C, D, E, F, G] // ...
case class Route8[A, B, C, D, E, F, G, H] // ...
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      hlistFunction2((a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b)))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      hlistFunction2((a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b)))


    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.url(1 :: 2 :: HNil)    // => “/add/1/to/2”

Calculator.repeat(“abc” :: 2 :: HNil) // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))


    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))


    }

}


Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
trait PimpedPath {

  type Fn
  type Rt <: Route[_]

  def >>(fn: Fn)(implicit site: Site): Rt

}
```

```scala
trait PimpedPath {

  type Fn
  type Rt <: Route[_]

  def >>(fn: Fn)(implicit site: Site): Rt

}

implicit def pimpPath2[A, B](
  path: Path { type Result = HCons[A, HCons[B, HNil]] }
) = new PimpedPath {

  type Fn = (A, B) => Response
  type Rt <: Route2[A, B]

  def >>(fn: Fn)(implicit site: Site) =
    Route2(site, path, fn)

}
```

```scala
implicit def pimpPath2[A, B](...) // ...
```

```scala
implicit def pimpPath0](...) // ...
implicit def pimpPath1[A](...) // ...
implicit def pimpPath2[A, B](...) // ...
implicit def pimpPath3[A, B, C](...) // ...
implicit def pimpPath4[A, B, C, D](...) // ...
implicit def pimpPath5[A, B, C, D, E](...) // ...
implicit def pimpPath6[A, B, C, D, E, F](...) // ...
implicit def pimpPath7[A, B, C, D, E, F, G](...) // ...
implicit def pimpPath8[A, B, C, D, E, F, G, H](...) // ...
implicit def pimpPath9[A, B, C, D, E, F, G, H, I](...) // ...
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))

    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
            ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
            ("repeat" :/: StringArg :/:
                 IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    pimpPath2("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    pimpPath2("repeat" :/: StringArg :/:
                  IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))
    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```

```scala
object Calculator extends Site {

  val add =
    ("add" :/: IntArg :/: "to" :/: IntArg :/: end) >> {
      (a: Int, b: Int) =>
        Response("%s + %s = %s".format(a, b, a + b))
    }

  val repeat =
    ("repeat" :/: StringArg :/: IntArg :/: "times" :/: end) >> {
      (a: String, b: Int) =>
        Response("%s * %s = %s".format(a, b, a * b))

    }

}
```

```scala
Calculator.add.url(1, 2)              // => “/add/1/to/2”

Calculator.repeat(“abc”, 2)           // => Response(“...”)
```