from **Scala** monadic effects to **Unison** algebraic effects

Introduction to **Unison**'s **algebraic effects** (**abilities**)

go from a small **Scala** program based on the **Option monad**

to a **Unison** program based on the **Abort ability**

\- inspired by, and part based on, a talk by **Runar Bjarnason** -

Scala

unison

```
-- Scala
for {
  a ← x
  b ← y
  c ← z
} yield f(a, b, c)

-- Unison
f x y z
```

Runar Bjarnason
@runarorama

In **Unison** you just say **f x y z** and it will figure that out. It will do the **pulling out**. It will do all the **effects**.

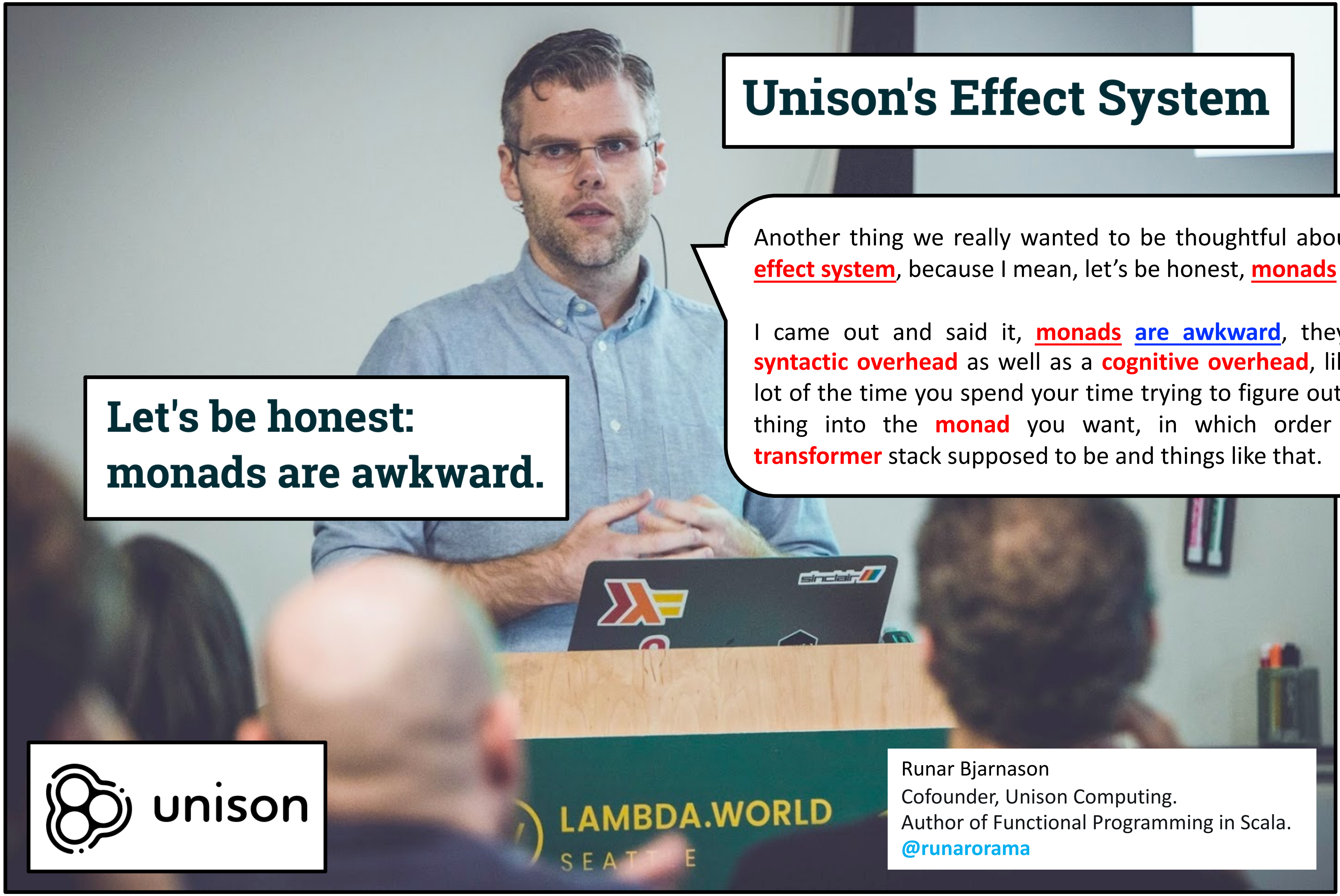
by  @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

We start off by looking at how **Runar Bjarnason** explains **Unison**'s **effect system** in his talk **Introduction to the Unison programming language**.

@philip_schwarz

Unison's Effect System



Let's be honest: monads are awkward.

Another thing we really wanted to be thoughtful about was **unison**'s **effect system**, because I mean, let's be honest, **monads are awkward**.

I came out and said it, **monads are awkward**, they come with a **syntactic overhead** as well as a **cognitive overhead**, like, you know, a lot of the time you spend your time trying to figure out how to lift this thing into the **monad** you want, in which order is my **monad transformer** stack supposed to be and things like that.

unison

Runar Bjarnason
Cofounder, Unison Computing.
Author of Functional Programming in Scala.
@runarorama

YouTube Lambda World 2018 - Introduction to the Unison programming language - Rúnar Bjarnason

So **Unison** uses what's sometimes known as **algebraic effects**. We modeled our **effect system** in a language called **Frank**, which is detailed in this paper, which is called **Do Be Do Be Do**, by **Sam Lindley**, **Conor McBride** and **Craig McLaughlin**, and **Frank** calls these **abilities**, rather than **effects**, and so we do that, we call them **abilities**.

Do Be Do Be Do

Sam Lindley, Conor McBride, Craig McLaughlin

https://arxiv.org/abs/1611.09259

Runar Bjarnason
@runarorama

Abilities

So here is a simple **State** ability.

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

This is the ability to **put** and **get** some global **state** of type **s**. **Abilities** are introduced with the **ability** keyword and this defines two functions, **put** and **get**.

put takes some **state** of type **s** and it **returns unit with the State ability attached to it**, and then **get will give you that s**, **given that you have the State ability**.

When we see a thing like this in curly braces, it means this requires that ability. So **put requires the State ability and get also requires the State ability**.

So this is very similar to an **Algebraic Data Type** where **you are defining the type State**, this **ability type**, and **these are the constructors of the type**: **put** and **get**.

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

So for example we can write effectful functions push and pop on a global stack.

```
pop : '{State [a]} Optional a
pop = '(stack = get
        put (drop 1 stack)
        head stack)

push : a → {State [a]} ()
push a = put (cons a get)
```

So given that the state is a stack, then we have the ability to manipulate some state that is a list of as: we can pop and push.

So note that there is no monadic plumbing here. These are just code blocks.

And so to pop, we get the stack, we drop one element from the stack, we put that and then we get the head of the stack. So that's pop. And then push, we just say, cons a onto the front of whatever we get, and put that.

Runar Bjarnason
@runarorama

The reason why the pop is quoted is that only computations can have effects, not values. So once you have computed a value, you can no longer have effects. So the quoting is just a nullary function that returns whatever this evaluates to.

There is no applicative syntax or anything like that, because we are actually overloading the function application syntax. So in unison applicative programming is the default. We chose that as a design constraint.

Applicative programming is the default

The types will ensure that you can’t go wrong here, that you are not talking about the wrong thing.

```
-- Scala
for {
  a ← x
  b ← y
  c ← z
} yield f(a, b, c)

-- Unison
f x y z
```

So for example whereas in **Scala** you might say **a** comes from **x**, **b** comes from **y** and then **c** comes from **z**, and then you want to do **f** of **a**, **b** and **c**.

In Unison you just say f x y z and it will figure that out. It will do the pulling out. It will do all the effects.

Runar Bjarnason
@runarorama

So whereas in **Haskell** you might have to say **x bind** lambda of **f** of **a** and then **bind g**, in **Unison** you just say **g** of **f** of **x**.

```
-- Haskell
x >>= (\a → f a >>= g)

-- Unison
g (f x)
```

So that’s kind of nice, there is a **low syntactic overhead** to this and there is a **low cognitive overhead** to this, for the programmer.

```
stackProgram : '{State [Nat]} ()
stackProgram =
  '( a = pop
     if a == 5
       then push 5
       else
         push 3
         push 8 )
```

So the programmer can just use our **pop** and **push** and write a little program that **pushes** and **pops** the **stack** using our **State** **ability**.

So given that we have the **ability** to manipulate some **state** of type **list** of **Nat**, we can write a **stack** program.

a is the **head** of the **stack**, we **pop** the **stack** and now we have **mutated** the **stack** and then if **a** is five then **push** it back, otherwise **push** 3 and 8.

So this looks like a little imperative program but it is actually a purely functional program.

There are **no side effects here** but there is **also no visible effect plumbing**.

Runar Bjarnason @runarorama

```
state : s → '({State s} a) → a
state s c =
  h s e = case e of
    { State.get → k } →
      handle h s in k s
    { State.put s → k } →
      handle h s in k ()
    { a } → a
  handle h s in !c

runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

So then **to handle the State ability**, to make it actually do something, **we write a handler using a handle keyword**.

This here is a **pure handler** for the **State ability** and we can use that **handler**, at the bottom, the **runStack** thing uses that **handler** to run the **stackProgram** with some initial **state** which is [5,4,3,2,1].

Normally this kind of stuff would be hidden away in library code. Most programmers will not be writing their own **handlers** but if you have your own set of **abilities**, you'll be able to write your **handlers**.

Runar Bjarnason @runarorama

```
state : s → '({State s} a) → a
state s c =
  h s e = case e of
    { State.get → k } →
      handle h s in k s
    { State.put s → k } →
      handle h s in k ()
    { a } → a
  handle h s in !c

runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

So here the definition of state, the expression here at the bottom is like handle h of s in bang c, where the exclamation sign means force this computation. c is some quoted computation, you can see that it is quoted in the type, it is something of type {State s} a, and then I am saying, force that, actually evaluate it, but handle using the handler h, or h of s, where s is the initial state coming in, it is that [5,4,3,2,1] thing.

And then the definition of h is just above and it proceeds by pattern matching on the constructors of the ability.

Runar Bjarnason @runarorama

```
state : s → '({State s} a) → a
state s c =
  h s e = case e of
    { State.get → k } →
      handle h s in k s
    { State.put s → k } →
      handle h s in k ()
    { a } → a
  handle h s in !c

runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

Runar Bjarnason @runarorama

If the call was to a get, then we end up in that case and what we get out of that pattern is k, a continuation for the program, the rest of the program, and what is expected is that I pass the current state to k, that is we allow the program to continue with the current state, so if there is a get then I call k of s and this is a recursive definition, I keep trying to handle if there is any more state manipulation going on, it is actually calling the handler again, because k of s might also need access to the state ability.

And then to put, we get a state that somebody wanted to put and we get the continuation of the program and we say well, handle that using the state and then continue by passing the unit to the continuation.

And then in the pure case, when there is no effect, we just return the value that we ended up with.

Runar Bjarnason @runarorama

The next slide has all of the **state** code shown by **Runar**.

@philip_schwarz

```
ability State s where
  put : s → {State s} ()
  get : {State s} s
```

```
pop : '{State [a]} Optional a
pop = '(stack = get
        put (drop 1 stack)
        head stack)

push : a → {State [a]} ()
push a = put (cons a get)
```

```
state : s → '({State s} a) → a
state s c =
  h s e = case e of
    { State.get → k } →
      handle h s in k s
    { State.put s → k } →
      handle h s in k ()
    { a } → a
  handle h s in !c
```

```
stackProgram : '{State [Nat]} ()
stackProgram =
  '( a = pop
     if a == 5
       then push 5
       else
         push 3
         push 8 )
```

```
runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

unison

When I went to run that code, I made the following changes to it:

- I updated it to reflect some minor changes to the **Unison** language which have occurred since **Runar** gave the talk.
- Since the **pop** function returns **Optional a**, I changed **stackProgram** so that it doesn't expect **pop** to return an **a**.
- Since **runStack** returns a **stack**, i.e. a list of numbers, I changed **stackProgram** to also return a **stack**.
- I changed a bit the pushing and popping that **stackProgram** does, and added **automated tests** to visualise the effect of that logic on a **stack**.
- Since the **pop** function returns a **quoted computation**, I prefixed invocations of **pop** with the **exclamations sign**, to **force the execution** of the **computations**.
- I prefixed usages of **put** and **get** with **State**.
- I added the **List.head** function that **pop** uses

See the next slide for the resulting code

```
ability State s where
  put : s -> {State s} ()
  get : {State s} s
```

```
pop : '{State [a]} (Optional a)
pop = 'let
  stack = State.get
  State.put (drop 1 stack)
  head stack
push : a -> {State [a]} ()
push a = State.put (cons a State.get)
```

```
state : s -> '({State s} a) -> a
state s c =
  h s cases
    { State.get -> k } ->
      handle k s with h s
    { State.put s -> k } ->
      handle k () with h s
    { a } -> a
  handle !c with h s
```

```
List.head : [a] -> Optional a
List.head a = List.at 0 a
use List head
```

unison

```
stackProgram : '{State [Nat]} [Nat]
stackProgram =
  'let top = !pop
       match top with
         None ->
           push 0
           push 1
           push 2
         Some 5 ->
           !pop
           push 5
         Some n ->
           !pop
           !pop
           push n
           push (n + n)
       State.get
```

```
test> topIsFive =
      check(state [5,4,3,2,1] stackProgram == [5,3,2,1])

test> topIsNotFive =
      check(state [6,5,4,3,2,1] stackProgram == [12,6,3,2,1])

test> topIsMissing =
      check(state [] stackProgram == [2,1,0])
```

```
runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

```
> runStack
  ⧩
  [5, 3, 2, 1]
```

To help understand how the **state** function works, I made the following changes to it:

- make the type of the **h** function explicit
- rename the **h** function to **handler**
- rename **c** to **computation**
- rename **k** to **continuation**
- break ‘each **handle** … **with** …’ line into two lines

The next slide shows the **state** function before and after the changes and the slide after that shows the whole code again after the changes to the **state** function.

```
state : s -> '({State s} a) -> a
state s c =
  h s cases
    { State.get -> k } ->
      handle k s with h s
    { State.put s -> k } ->
      handle k () with h s
    { a } -> a
  handle !c with h s
```

```
state : s -> '({State s} a) -> a
state s computation =
  handler : s -> Request {State s} a -> a
  handler s cases
    { State.get -> continuation } ->
      handle continuation s
        with handler s
    { State.put s -> continuation } ->
      handle continuation ()
        with handler s
    { a } -> a
  handle !computation
    with handler s
```

@philip_schwarz

In https://www.unisonweb.org/docs/language-reference we read the following:

> .base **Request** is the constructor of requests for **abilities**. A type **Request A T** is the type of values received by **ability handlers** for the ability **A** where the current continuation requires a value of type **T**.

So on the right we see the **state handler** function taking first a **state**, and then a `Request {State s} a`, i.e. a request for the **State ability** where the continuation requires a value of type **a**.

```
ability State s where
  put : s -> {State s} ()
  get : {State s} s
```

```
pop : '{State [a]} (Optional a)
pop = 'let
  stack = State.get
  State.put (drop 1 stack)
  head stack
push : a -> {State [a]} ()
push a = State.put (cons a State.get)
```

```
state : s -> '({State s} a) -> a
state s computation =
  handler : s -> Request {State s} a -> a
  handler s cases
    { State.get -> continuation } ->
      handle continuation s
        with handler s
    { State.put s -> continuation } ->
      handle continuation ()
        with handler s
    { a } -> a
  handle !computation
    with handler s
```

```
List.head : [a] -> Optional a
List.head a = List.at 0 a
use List head
```

```
stackProgram : '{State [Nat]} [Nat]
stackProgram =
  'let top = !pop
       match top with
         None ->
           push 0
           push 1
           push 2
         Some 5 ->
           !pop
           push 5
         Some n ->
           !pop
           !pop
           push n
           push (n + n)
       State.get
```

```
test> topIsFive =
      check(state [5,4,3,2,1] stackProgram == [5,3,2,1])

test> topIsNotFive =
      check(state [6,5,4,3,2,1] stackProgram == [12,6,3,2,1])

test> topIsMissing =
      check(state [] stackProgram == [2,1,0])
```

```
runStack : [Nat]
runStack = state [5,4,3,2,1] stackProgram
```

```
> runStack
  ⧩
  [5, 3, 2, 1]
```

Now back to **Runar**'s talk for one more fact about functional effects in **Unison**.

And yes, you can still use **monads**, if you want.

You don't ***have to*** use this **ability** stuff.

You can still use **monads** and it will work just fine.

Runar Bjarnason
@runarorama

Yes, you can still use monads.

Earlier on **Runar** showed us a comparison between a **Scala monadic for comprehension** and a **Unison** plain function invocation that instead relied on an **ability**. He also showed us a comparison between a **Haskell** expression using the **bind** function (**flatMap** in **Scala**) and a **Unison** plain function invocation that again relied on an **ability**.

```
-- Scala
for {
  a ← x
  b ← y
  c ← z
} yield f(a, b, c)

-- Unison
f x y z
```

```
-- Haskell
x >>= (\a → f a >>= g)

-- Unison
g (f x)
```

In the rest of this slide deck, we are going to do two things:

- Firstly, we are going to look at an example of how **functional effects** look like in **Unison** when we use **monadic effects** rather than **algebraic effects**. i.e we are going to use a **monad** rather than an **ability**. We are going to do that by starting with a very small **Scala** program that uses a **monad** and then translating the program into the **Unison** equivalent.
- Secondly, we want to see another **Unison** example of implementing a **functional effect** using an **ability**, so we are going to take the above **Unison** program and convert it so that it uses an **ability** rather than a **monad**.

In the process we'll be making the following comparisons:

```
-- Scala program that
-- uses the Option monad
x flatMap { a =>
  y flatMap { b =>
    z map { c =>
      f(a,b,c)
    }
  }
}

-- Unison program that
-- uses the Optional monad
???
```

```
-- Scala program that
-- uses the Option monad
for {
  a <- x
  b <- y
  c <- z
} yield f(a b c)

-- Unison program that
-- uses the Abort ability
???
```

@philip_schwarz

The **state** functional effect is not the easiest to understand, so to aid our understanding, the program we'll be looking at is simply going to do validation using the **functional effect** of **optionality**.

The **Scala** program that we'll be translating into **Unison**. Is on the next slide.

@philip_schwarz

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

def createPerson(name: String, surname: String, age: Int): Option[Person] =
  for {
    aName    <- validateName(name)
    aSurname <- validateSurname(surname)
    anAge    <- validateAge(age)
  } yield Person(aName, aSurname, anAge)
```

```scala
val people: String =
  potentialPeople
    .foldLeft("")(((text,person) => text + "\n" + toText(person)))

assert( people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone" )
```

```scala
def toText(option: Option[Person]): String =
  option match {
    case Some(person) => person.toString
    case None => "None"
  }
```

```scala
val potentialPeople = List(
  createPerson("Fred", "Smith", 35),
  createPerson(   "x", "Smith", 35),
  createPerson("Fred",      "", 35),
  createPerson("Fred", "Smith",  0)
)
```

```
println(people)
➔
Person(Fred,Smith,35)
None
None
None
```

```scala
def validateName(name: String): Option[String] =
  if (name.size > 1 && name.size < 15)
    Some(name)
  else None

def validateSurname(surname: String): Option[String] =
  if (surname.size > 1 && surname.size < 20)
    Some(surname)
  else None

def validateAge(age: Int): Option[Int] =
  if (age > 0 && age < 112)
    Some(age)
  else None
```

```scala
sealed trait Option[+A] {

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case Some(a) => f(a)
      case None    => None
    }

}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]
```

Let's begin by translating the validation functions. The Unison equivalent of Scala' s Option is the Optional type.

```scala
def validateName(name: String): Option[String] =
  if (name.size > 1 && name.size < 15)
    Some(name)
  else None

def validateSurname(surname: String): Option[String] =
  if (surname.size > 1 && surname.size < 20)
    Some(surname)
  else None

def validateAge(age: Int): Option[Int] =
  if (age > 0 && age < 112)
    Some(age)
  else None
```

```
validateName : Text -> Optional Text
validateName name =
  if (size name > 1) && (size name < 15)
  then Some name
  else None

validateSurname : Text -> Optional Text
validateSurname surname =
  if (size surname > 1) && (size surname < 20)
  then Some surname
  else None

validateAge : Nat -> Optional Nat
validateAge age =
  if (age > 0) && (age < 112)
  then Some age
  else None
```

Scala

unison

as a minor aside, if we were using the **Scala** built-in **Option** type then we would have the option of rewriting code like this

```scala
if (age > 0 && age < 112)
  Some(age)
else None
```

as follows

```scala
Option.when(age > 0 && age < 112)(age)
```

or alternatively as follows

```scala
Option.unless(age <= 0 || age > 112)(age)
```

Now that we have the validation functions in place, let's look at the translation of the **functional effect** of **optionality**.

On the left hand side we have a handrolled **Scala Option** with **map** defined in terms of **flatMap**, and on the right hand side we have the **Unison** predefined **Optional** type and its predefined **map** and **flatMap** functions.

```scala
sealed trait Option[+A] {

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case Some(a) => f(a)
      case None    => None
    }

}

case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]
```

```
type base.Optional a = None | Some a

base.Optional.map : (a -> b) -> Optional a -> Optional b
base.Optional.map f = cases
  None   -> None
  Some a -> Some (f a)

base.Optional.flatMap : (a -> Optional b) -> Optional a -> Optional b
base.Optional.flatMap f = cases
  None   -> None
  Some a -> f a

use .base.Optional map flatMap
```

Scala

unison

Now that we have the **map** and **flatMap** functions in place, let's look at the translation of the **Scala for comprehension** into **Unison**.

We are implementing the **functional effect** of **optionality** using a **monad**, so while in **Scala** we can use the **syntactic sugar** of a **for comprehension**, in **Unison** there is no equivalent of the **for comprehension** (AFAIK) and so we are having to use an explicit chain of **flatMap** and **map**.

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

def createPerson(name : String, surname: String, age: Int): Option[Person] =
  for {
    aName    <- validateName(name)
    aSurname <- validateSurname(surname)
    anAge    <- validateAge(age)
  } yield Person(aName, aSurname, anAge)
```

```
type Person = { name: Text, surname: Text, age: Nat }

use .base.Optional map flatMap

createPerson : Text -> Text -> Nat -> Optional Person
createPerson name surname age =
  flatMap (aName ->
    flatMap (aSurname ->
      map (anAge ->
        Person.Person aName aSurname anAge
      )(validateAge age)
    )(validateSurname surname)
  )(validateName name)
```

Scala

unison

Here is the same comparison as on the previous slide but with the **Scala** code explicitly using **map** and **flatMap**.

@philip_schwarz

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

def createPerson(name : String, surname: String, age: Int): Option[Person] =
  validateName(name) flatMap { aName =>
    validateSurname(surname) flatMap { aSurname =>
      validateAge(age) map { anAge =>
        Person(aName, aSurname, anAge)
      }
    }
  }
```

Scala

```
type Person = { name: Text, surname: Text, age: Nat }

use .base.Optional map flatMap

createPerson : Text -> Text -> Nat -> Optional Person
createPerson name surname age =
  flatMap (aName ->
    flatMap (aSurname ->
      map (anAge ->
        Person.Person aName aSurname anAge
      )(validateAge age)
    )(validateSurname surname)
  )(validateName name)
```

unison

```scala
val people: String =
  potentialPeople
    .foldLeft("")(((text,person) => text + "\n" + toText(person)))

assert(
 people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone"
)
```

```haskell
people : Text
people = foldl
           (text person -> text ++ "\n" ++ (toText person))
           ""
           potentialPeople

peopleTest = check (
  people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone"
)
```

```scala
val potentialPeople: List[Option[Person]] =
  List( createPerson("Fred", "Smith", 35),
        createPerson(   "x", "Smith", 35),
        createPerson("Fred",      "", 35),
        createPerson("Fred", "Smith",  0)  )
```

```haskell
potentialPeople: [Optional Person]
potentialPeople =
  [ (createPerson "Fred" "Smith" 35),
    (createPerson    "x" "Smith" 35),
    (createPerson "Fred"      "" 35),
    (createPerson "Fred" "Smith"  0) ]
```

```scala
def toText(option: Option[Person]): String =
  option match {
    case Some(person) => person.toString
    case None => "None"
  }
```

```haskell
toText : Optional Person -> Text
toText = cases
  Some person -> Person.toText person
  None        -> "None”

Person.toText : Person -> Text
Person.toText person =
  match person with Person.Person name surname age
    -> "Person(" ++ name ++ ","
                 ++ surname ++ ","
                 ++ Text.toText(age) ++ ")"
```

Here we translate the rest of the program.

See the next slide for the **Unison** translation of the whole **Scala** program.

@philip_schwarz

```
type Person = {
  name: Text, surname: Text, age: Nat
}

use .base.Optional map flatMap

createPerson : Text -> Text -> Nat -> Optional Person
createPerson name surname age =
  flatMap (aName ->
    flatMap (aSurname ->
      map (anAge ->
        Person.Person aName aSurname anAge
      )(validateAge age)
    )(validateSurname surname)
  )(validateName name)
```

```
validateName : Text -> Optional Text
validateName name =
  if (size name > 1) && (size name < 15)
  then Some name
  else None

validateSurname : Text -> Optional Text
validateSurname surname =
  if (size surname > 1) && (size surname < 20)
  then Some surname
  else None

validateAge : Nat -> Optional Nat
validateAge age =
  if (age > 0) && (age < 112)
  then Some age
  else None
```

```
people : Text
people = foldl
           (text person -> text ++ "\n" ++ (toText person))
           ""
           potentialPeople

peopleTest = check (people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone")
```

```
toText : Optional Person -> Text
toText = cases
  Some person -> Person.toText person
  None        -> "None"

Person.toText : Person -> Text
Person.toText person =
  match person with
    Person.Person name surname age
    -> "Person(" ++
         name ++ "," ++
         surname ++ "," ++
         Text.toText(age) ++ ")"
```

unison

```
potentialPeople: [Optional Person]
potentialPeople =
  [(createPerson "Fred" "Smith" 35),
   (createPerson    "x" "Smith" 35),
   (createPerson "Fred"      "" 35),
   (createPerson "Fred" "Smith"  0)]
```

```
type base.Optional a = None | Some a

base.Optional.map : (a -> b) -> Optional a -> Optional b
base.Optional.map f = cases
  None   -> None
  Some a -> Some (f a)

base.Optional.flatMap : (a -> Optional b) -> Optional a -> Optional b
base.Optional.flatMap f = cases
  None   -> None
  Some a -> f a
```

On the next slide we look at some simple automated tests for the **Unison** program.

```
test>                                  peopleTest = check (people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone")

test>                           validPersonAsText = check (Person.toText (Person.Person "Fred" "Smith" 35) == "Person(Fred,Smith,35)")

test>                                 validPerson = check (createPerson "Fred" "Smith" 35 == Some (Person.Person "Fred" "Smith" 35))

test>                noValidPersonWithInvalidName = check (createPerson "F" "Smith" 35 == None)
test>             noValidPersonWithInvalidSurname = check (createPerson "Fred" "" 35 == None)
test>                 noValidPersonWithInvalidAge = check (createPerson "Fred" "Smith" 200 == None)
test> noValidPersonWithInvalidNameSurnameAndAge = check (createPerson "" "S" 200 == None)

test>                                   validName = check (validateName "Fred" == Some "Fred")
test>                                validSurname = check (validateSurname "Smith" == Some "Smith")
test>                                    validAge = check (validateAge 35 == Some 35)

test>                               noInvalidName = check (validateName "" == None)
test>                            noInvalidSurname = check (validateSurname "X" == None)
test>                                noInvalidAge = check (validateAge 200 == None)
```

As we have seen, the **Unison** program currently implements the **functional effect** of **optionality** using the **Optional monad**.

What we are going to do next is improve that program, make it easier to understand, by changing it so that it implements the **effect** of **optionality** using an **ability** (**algebraic effect**) called **Abort**.

Let’s begin by looking at the **Abort** ability. Although it is a **predefined ability**, on this slide I have refactored the original a bit so that we can better compare it with the **State ability** that we saw earlier in **Runar**’s code.

In later slides I am going to revert to the **predefined** version of the **ability**, which being split into two functions, offers different advantages.

The **Abort ability** is much simpler than the **State ability**, that’s why I think it could be a good first example of using **abilities**.

```
ability State s where
  put : s -> {State s} ()
  get : {State s} s
```

```
ability Abort where
  abort : {Abort} a
```

```
state : s -> '({State s} a) -> a
state s computation =
  handler : s -> Request {State s} a -> a
  handler s cases
    { State.get -> continuation } ->
      handle continuation s
        with handler s
    { State.put s -> continuation } ->
      handle continuation ()
        with handler s
    { a } -> a
  handle !computation
    with handler s
```

```
Abort.toOptional : '{g, Abort} a -> {g} Optional a
Abort.toOptional computation =
  handler : Request {Abort} a -> Optional a
  handler = cases
    { a }              -> Some a
    { abort -> _ } -> None
  handle !computation
    with handler
```

To help understand how the **handler** of the **Abort ability** works, in the next slide we look at some relevant documentation from a similar **Abort ability** in the **Unison** language reference.
By the way, it looks like that **g** in the the signature of the toOptional function somehow caters for potentially multiple **abilities** being in play at the same time, but we’ll just ignore that aspect because it is out of scope for our purposes.

```
ability Abort where
  aborting : ()

-- Returns `a` immediately if the
-- program `e` calls `abort`
abortHandler : a -> Request Abort a -> a
abortHandler a = cases
   { Abort.aborting -> _ } -> a
   { x } -> x

p : Nat
p = handle
      x = 4
      Abort.aborting
      x + 2
    with abortHandler 0
```

A **handler** can choose to call the **continuation** or not, or to call it multiple times. For example, a **handler** can ignore the **continuation** in order to handle an **ability** that **aborts** the execution of the program.

The program **p** evaluates to 0. If we remove the **Abort.aborting** call, it evaluates to 6.

Note that although the **ability** constructor is given the signature **aborting** : (), its actual type is {**Abort**} ().

The pattern { **Abort.aborting** -> _ } matches when the **Abort.aborting** call in **p** occurs. This pattern ignores its **continuation** since it will not invoke it (which is how it **aborts** the program). The continuation at this point is the expression _ -> x + 2.

The pattern { x } matches the case where the computation is **pure** (makes no further requests for the **Abort** ability and the **continuation** is **empty**). A pattern match on a **Request** is not complete unless this case is handled.

from https://www.unisonweb.org/docs/language-reference

As I said on the previous slide, while the above **Abort ability** is similar to the one we are going to use, it is not identical. e.g. this **handler** returns an **a** rather than an **Optional a**. The reason why we are looking at this example is because the patterns in the **handler** are identical and the above explanations are also useful for the **Abort** ability that we are going to use.

unison

unison

So here on the left are the **map** and **flatMap** functions that the program currently uses to implement the **functional effect** of **optionality** and on the right is the predefined **Abort** ability that the program is now going to use instead.

@philip_schwarz

```
type base.Optional a = None | Some a

base.Optional.map : (a -> b) -> Optional a -> Optional b
base.Optional.map f = cases
  None   -> None
  Some a -> Some (f a)

base.Optional.flatMap : (a -> Optional b) -> Optional a -> Optional b
base.Optional.flatMap f = cases
  None   -> None
  Some a -> f a
```

```
type base.Optional a = None | Some a

ability Abort where
  abort : {Abort} a

Abort.toOptional.handler : Request {Abort} a -> Optional a
Abort.toOptional.handler = cases
  { a }             -> Some a
  { abort -> _ } -> None

Abort.toOptional : '{g, Abort} a -> {g} Optional a
Abort.toOptional a =
  handle !a with toOptional.handler
```

unison

Here we refactor the validation functions and on the next slide we refactor the majority of the rest of the program, leaving the most interesting bit of refactoring for the slide after that.

```
validateName : Text -> Optional Text
validateName name =
  if (size name > 1) && (size name < 15)
  then Some name
  else None

validateSurname : Text -> Optional Text
validateSurname surname =
  if (size surname > 1) && (size surname < 20)
  then Some surname
  else None

validateAge : Nat -> Optional Nat
validateAge age =
  if (age > 0) && (age < 112)
  then Some age
  else None
```

```
validateName : Text -> { Abort } Text
validateName name =
  if (size name > 1) && (size name < 15)
  then name
  else abort

validateSurname : Text -> { Abort } Text
validateSurname surname =
  if (size surname > 1) && (size surname < 20)
  then surname
  else abort

validateAge : Nat -> { Abort } Nat
validateAge age =
  if (age > 0) && (age < 112)
  then age
  else abort
```

```
people : Text
people =
 foldl (text person -> text ++ "\n" ++ (toText person))
       ""
       potentialPeople

peopleTest = check (
  people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone”
)
```

```
potentialPeople: [Optional Person]
potentialPeople =
  [ (createPerson "Fred" "Smith" 35),
    (createPerson    "x" "Smith" 35),
    (createPerson "Fred"      "" 35),
    (createPerson "Fred" "Smith"  0) ]
```

```
toText : Optional Person -> Text
toText = cases
  Some person -> Person.toText person
  None        -> "None”

Person.toText : Person -> {} Text
Person.toText person =
  match person with Person.Person name surname age
    -> "Person(" ++ name ++ ","
                 ++ surname ++ ","
                 ++ Text.toText(age) ++ ")"
```

```
people : Text
people =
  foldl (text person -> text ++ "\n" ++ toText (toOptional person))
        ""
        potentialPeople

peopleTest = check (
  people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone”
)
```

```
potentialPeople : ['{Abort} Person]
potentialPeople =
  [ '(createPerson "Fred" "Smith" 35),
    '(createPerson    "x" "Smith" 35),
    '(createPerson "Fred"      "" 35),
    '(createPerson "Fred" "Smith"  0) ]
```

```
toText : Optional Person -> Text
toText = cases
  Some person -> Person.toText person
  None        -> "None”

Person.toText : Person -> {} Text
Person.toText person =
  match person with Person.Person name surname age
    -> "Person(" ++ name ++ ","
                 ++ surname ++ ","
                 ++ Text.toText(age) ++ ")"
```

unison

And now the most interesting bit of the refactoring.

See how much simpler the **createPerson** function becomes when the **functional effect** of **optionality** is implemented not using a **monad** but using an **ability** and its **handler**.

```
type Person = { name: Text, surname: Text, age: Nat }

createPerson : Text -> Text -> Nat -> Optional Person
createPerson name surname age =
  flatMap (aName ->
    flatMap (aSurname ->
      map (anAge ->
        Person.Person aName aSurname anAge
      )(validateAge age)
    )(validateSurname surname)
  )(validateName name)
```

```
type Person = { name: Text, surname: Text, age: Nat }

createPerson : Text -> Text -> Nat -> { Abort } Person
createPerson name surname age =
  Person.Person
    (validateName name)
    (validateSurname surname)
    (validateAge age)
```

This new version of the **createPerson** function, which uses an **ability** (and its associated **handler**) is not only an improvement over the version that uses a **monad** but also over the **Scala** version that itself improves on explicit **monadic code** by using a **for comprehension**.

```
-- Scala
for {
  a ← x
  b ← y
  c ← z
} yield f(a, b, c)

-- Unison
f x y z
```

In Unison you just say **f x y z** and it will figure that out. It will do the **pulling out**. It will do all the **effects**.

Runar Bjarnason
@runarorama

```
-- Scala
for {
  aName    <- validateName(name)
  aSurname <- validateSurname(surname)
  anAge    <- validateAge(age)
} yield Person(aName, aSurname, anAge)

-- Unison
Person.Person
  (validateName name)
  (validateSurname surname)
  (validateAge age)
```

See the next slide for all the code of the refactored **Unison** program. See the subsequent slide for associated automated tests.

@philip_schwarz

```
type Person = { name: Text, surname: Text, age: Nat }

createPerson : Text -> Text -> Nat -> { Abort } Person
createPerson name surname age =
  Person.Person
    (validateName name)
    (validateSurname surname)
    (validateAge age)
```

```
validateName : Text -> { Abort } Text
validateName name =
  if (size name > 1) && (size name < 15)
  then name
  else abort

validateSurname : Text -> { Abort } Text
validateSurname surname =
  if (size surname > 1) && (size surname < 20)
  then surname
  else abort

validateAge : Nat -> { Abort } Nat
validateAge age =
  if (age > 0) && (age < 112)
  then age
  else abort
```

```
potentialPeople: ['{Abort} Person]
potentialPeople =
  [ '(createPerson "Fred" "Smith" 35),
    '(createPerson    "x" "Smith" 35),
    '(createPerson "Fred"      "" 35),
    '(createPerson "Fred" "Smith"  0) ]
```

```
people : Text
people =
 foldl (text person -> text ++ "\n" ++ toText (toOptional person))

       ""
       potentialPeople

peopleTest = check (people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone”)
```

```
toText : Optional Person -> Text
toText = cases
  Some person -> Person.toText person
  None        -> "None”

Person.toText : Person -> {} Text
Person.toText person =
  match person with Person.Person name surname age
    -> "Person(" ++ name ++ ","
                 ++ surname ++ ","
                 ++ Text.toText(age) ++ ")"
```

```
type base.Optional a = None | Some a

ability Abort where
  abort : {Abort} a

Abort.toOptional.handler : Request {Abort} a -> Optional a
Abort.toOptional.handler = cases
  { a }             -> Some a
  { abort -> _ } -> None

Abort.toOptional : '{g, Abort} a -> {g} Optional a
Abort.toOptional a =
  handle !a with toOptional.handler
```

```
test>                                      peopleTest = check (people == "\nPerson(Fred,Smith,35)\nNone\nNone\nNone")

test>                               validPersonAsText = check (Person.toText (Person.Person "Fred" "Smith" 35) == "Person(Fred,Smith,35)")

test>                               createValidPerson = check ((toOptional '(createPerson "Fred" "Smith" 35)) == Some((Person.Person "Fred" "Smith" 35)))


test>                             abortAsOptionIsNone = check (toOptional 'abort == None)

test>                   abortExpressionAsOptionIsNone = check (toOptional '(if false then "abc" else abort) == None)

test>                nonAbortExpressionAsOptionIsSome = check (toOptional '(if true then "abc" else abort) == Some "abc")


test>                  notCreatePersonWithInvalidName = check (toOptional('(createPerson "F" "Smith" 35)) == toOptional('abort))

test>               notCreatePersonWithInvalidSurname = check (toOptional('(createPerson "Fred" "" 35)) == toOptional('abort))

test>                   notCreatePersonWithInvalidAge = check (toOptional('(createPerson "Fred" "Smith" 200)) == toOptional('abort))


test>             personWithInvalidNameAsOptionIsNone = check (toOptional '(createPerson "F" "Smith" 35) == None)

test>          personWithInvalidSurnameAsOptionIsNone = check (toOptional '(createPerson "Fred" "" 35) == None)

test>              personWithInvalidAgeAsOptionIsNone = check (toOptional '(createPerson "Fred" "Smith" 200) == None)

test>        personWithAllInvalidFieldsAsOptionIsNone = check (toOptional '(createPerson "" "S" 200) == None)


test>                       invalidNameAsOptionIsNone = check (toOptional '(validateName "") == None)

test>                    invalidSurnameAsOptionIsNone = check (toOptional '(validateSurname "X") == None)

test>                        invalidAgeAsOptionIsNone = check (toOptional '(validateAge 200) == None)


test>                         validNameAsOptionIsSome = check (toOptional '(validateName "Fred") == Some "Fred")

test>                      validSurnameAsOptionIsSome = check (toOptional '(validateSurname "Smith") == Some "Smith")

test>                          validAgeAsOptionIsSome = check (toOptional '(validateAge 35) == Some 35)
```


That's all Folks!