NOKIA Bell Labs

Verifying state machine transitions with Scala types

Lambda Days

2019-02-22

Daniel Urban

Research Engineer

daniel.urban@nokia-bell-labs.com

durban

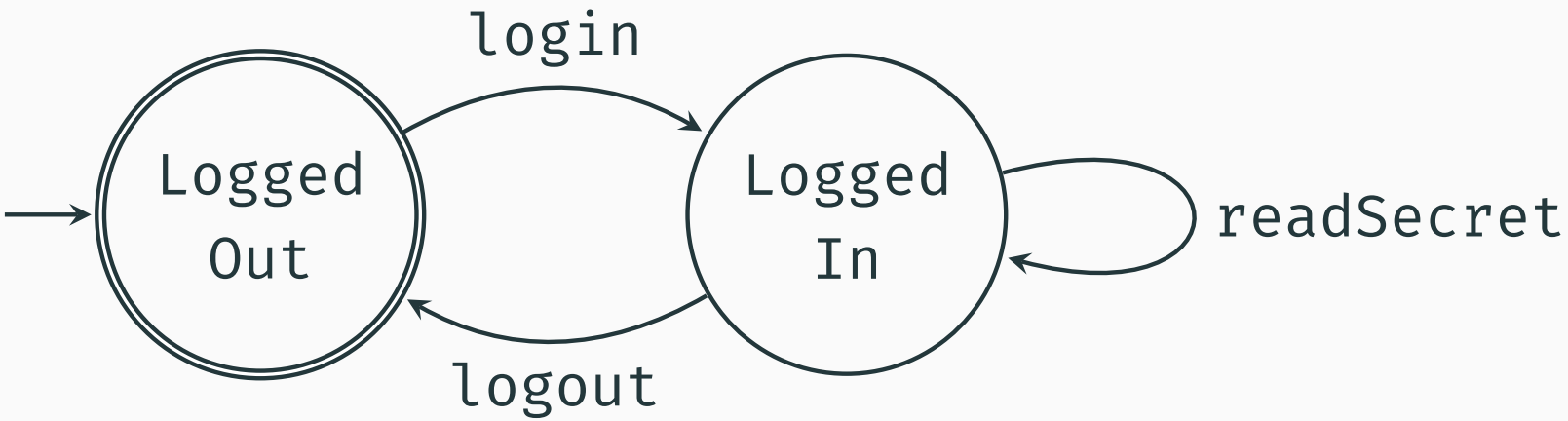
isorecursive

urban.dani@gmail.com

36A8 2002 483A 4CBF A5F8
DF6F 48B2 9573 BF19 7B13

API example

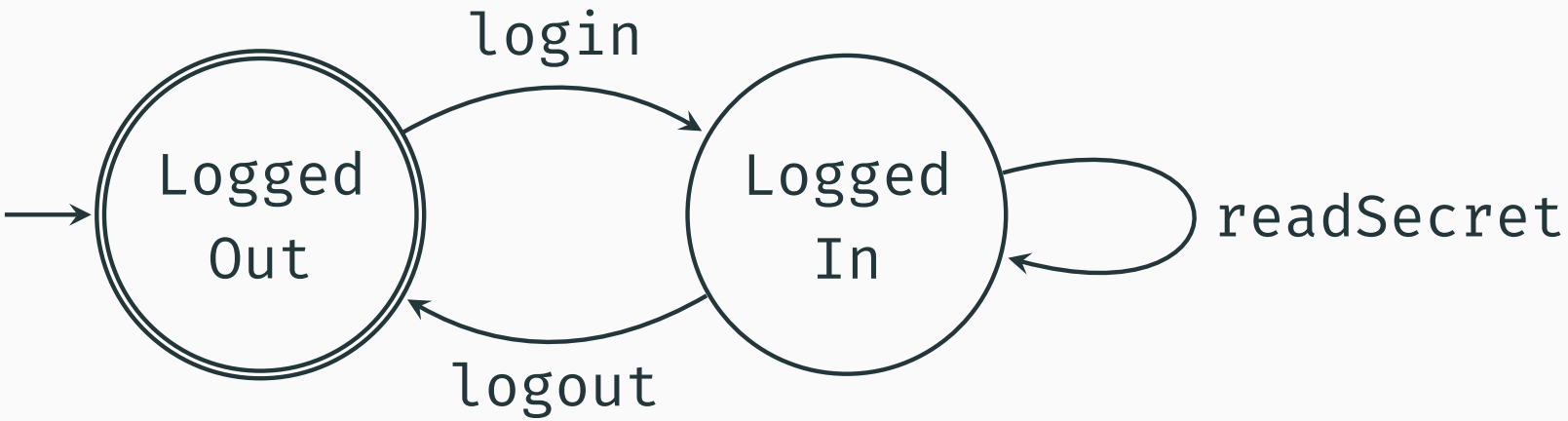
- Simple API for handling users:*



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

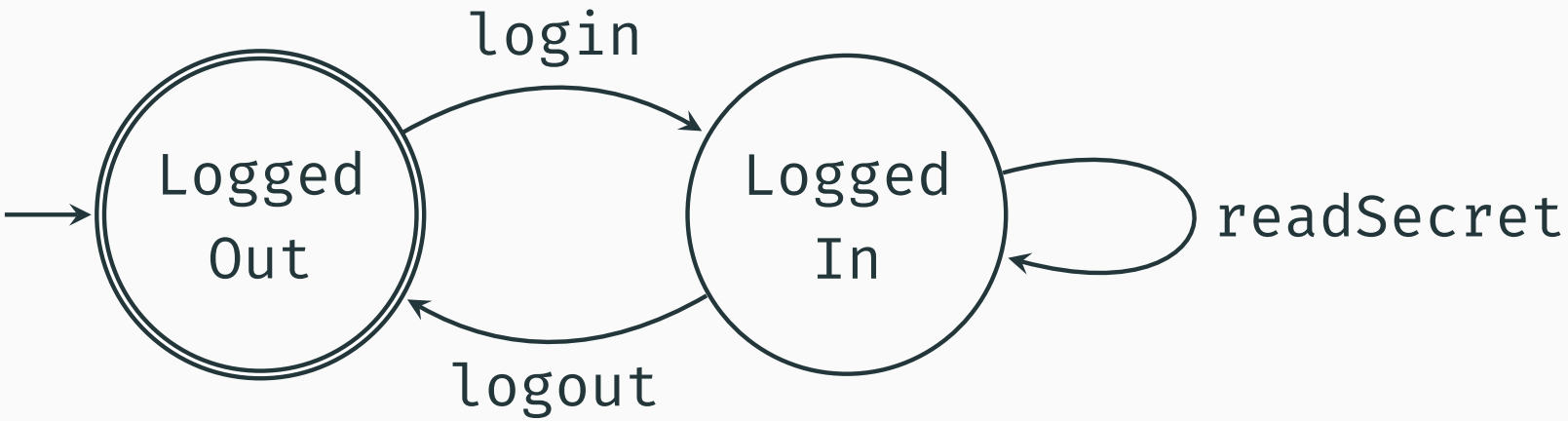
- Simple API for handling users:*
- Possible operations:



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

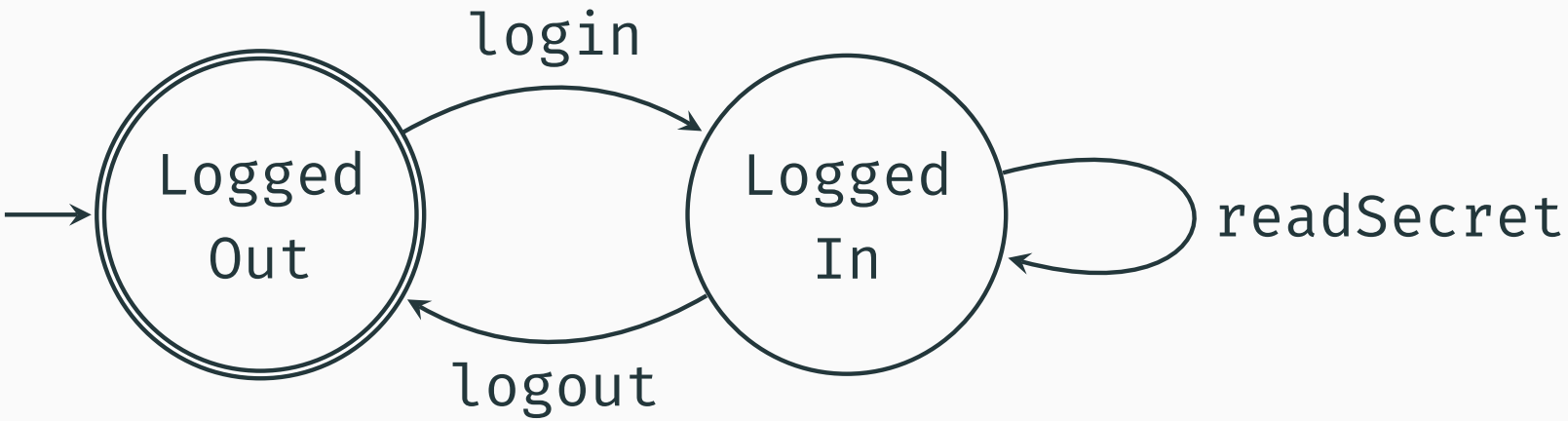
- Simple API for handling users:*
- Possible operations:
  - logging in



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

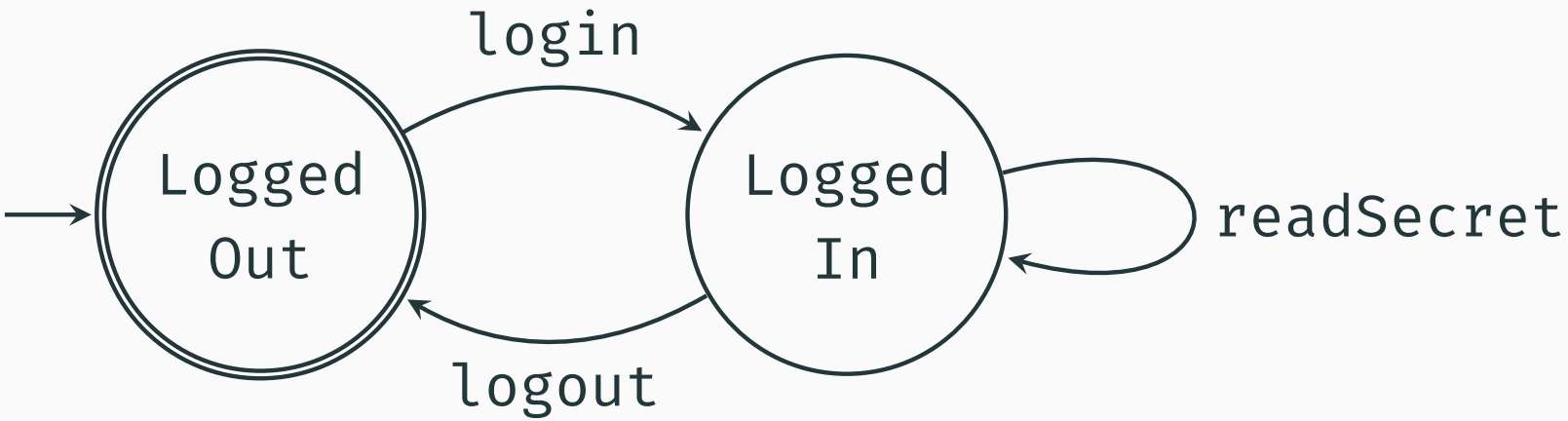
- Simple API for handling users:*
- Possible operations:
  - logging in
  - reading some “secret” data



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

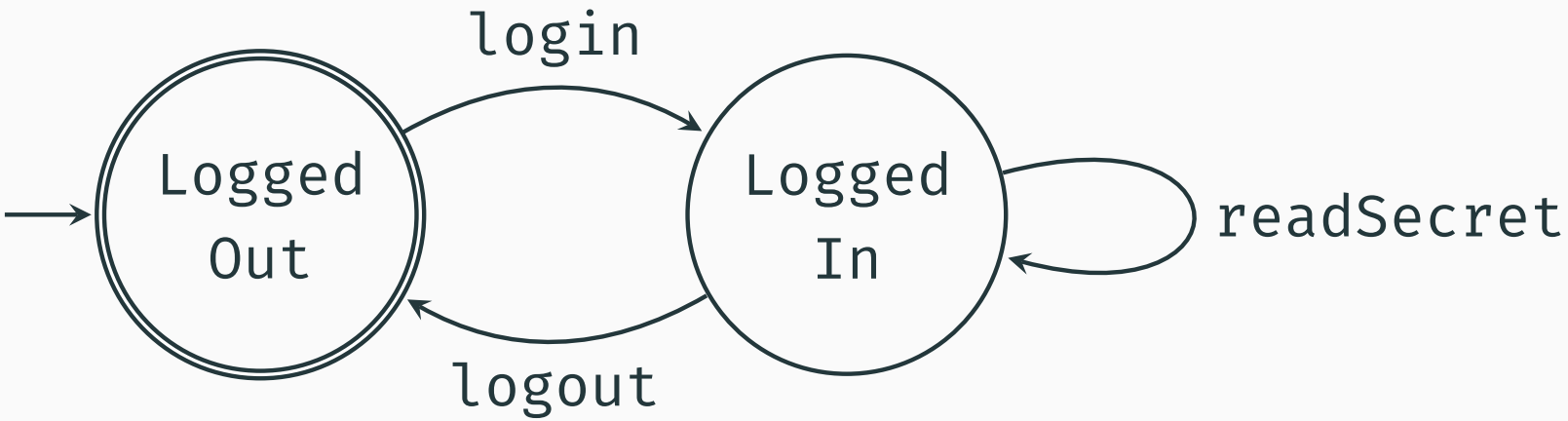
- Simple API for handling users:*
- Possible operations:
  - logging in
  - reading some “secret” data
  - logging out



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

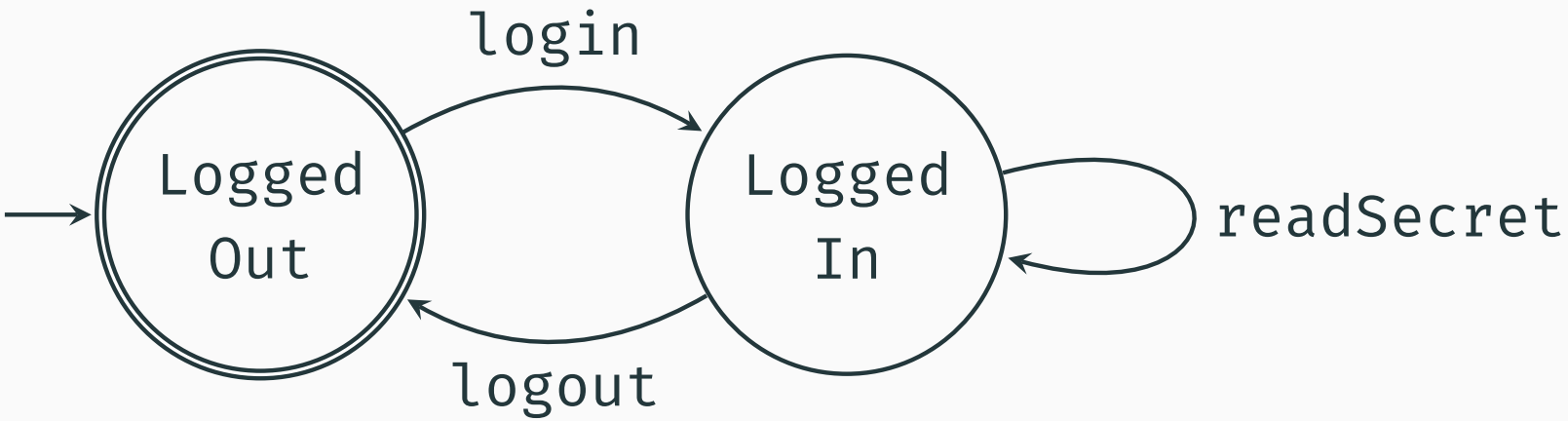
- Simple API for handling users:*
- Possible operations:
  - logging in
  - reading some “secret” data
  - logging out
- Constraints on the operations:



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

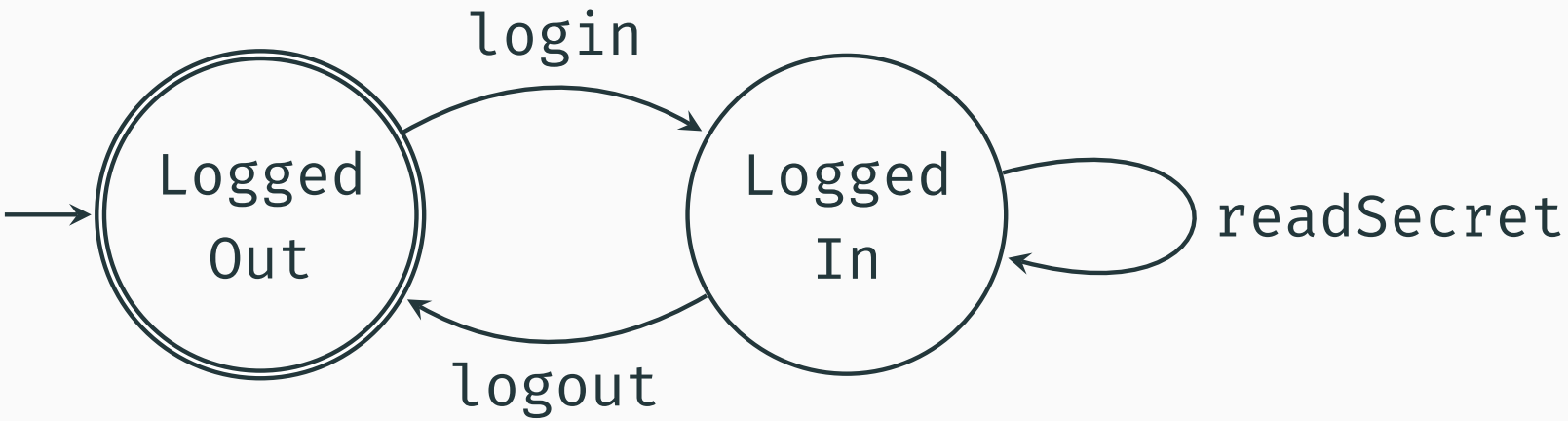
- Simple API for handling users:*
- Possible operations:
  - logging in
  - reading some “secret” data
  - logging out
- Constraints on the operations:
  - reading the secret is allowed *only* in the `LoggedIn` state



*From `https://www.idris-lang.org/drafts/sms.pdf`

API example

- Simple API for handling users:*
- Possible operations:
  - logging in
  - reading some “secret” data
  - logging out
- Constraints on the operations:
  - reading the secret is allowed *only* in the `LoggedIn` state
  - …



*From `https://www.idris-lang.org/drafts/sms.pdf`

Using a free monad

```scala
trait UserApi {
  def login(c: Credentials): Free[UserOp, Unit]
  def readSecret: Free[UserOp, String]
  def logout: Free[UserOp, Unit]
}
```

```scala
val myProgram: Free[UserOp, String] = for {
  _ ← UserApi.login(myCredentials)
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret

val interpreter: UserOp ~> IO = ??? // ...

val myIO: IO[String] =
  myProgram.foldMap[IO](interpreter)
```

Using a free monad

```scala
trait UserApi {
  def login(c: Credentials): Free[UserOp, Unit]
  def readSecret: Free[UserOp, String]
  def logout: Free[UserOp, Unit]
}

val myProgram: Free[UserOp, String] = for {
  _ ← UserApi.login(myCredentials)
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret

val interpreter: UserOp ~> IO = ??? // ...

val myIO: IO[String] =
  myProgram.foldMap[IO](interpreter)
```

Using a free monad

```scala
trait UserApi {
  def login(c: Credentials): Free[UserOp, Unit]
  def readSecret: Free[UserOp, String]
  def logout: Free[UserOp, Unit]
}

val myProgram: Free[UserOp, String] = for {
  _ ← UserApi.login(myCredentials)
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret

val interpreter: UserOp ~> IO = ??? // ...

val myIO: IO[String] =
  myProgram.foldMap[IO](interpreter)
```

Using a free monad

```scala
trait UserApi {
  def login(c: Credentials): Free[UserOp, Unit]
  def readSecret: Free[UserOp, String]
  def logout: Free[UserOp, Unit]
}

val myProgram: Free[UserOp, String] = for {
  _ ← UserApi.login(myCredentials)
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret

val interpreter: UserOp ~> IO = ??? // ...

val myIO: IO[String] =
  myProgram.foldMap[IO](interpreter)
```

Problems with this API

```scala
val badProgram: Free[UserOp, String] = for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials)
} yield secret

val badIO: IO[String] =
  badProgram.foldMap[IO](interpreter)
```

Problems with this API

```
val badProgram: Free[UserOp, String] = for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials)
} yield secret

val badIO: IO[String] =
  badProgram.foldMap[IO](interpreter)
```

- We can interpret and run our bad program
  - but we'll probably get a *runtime* error

Problems with this API

```
val badProgram: Free[UserOp, String] = for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials)
} yield secret

val badIO: IO[String] =
  badProgram.foldMap[IO](interpreter)
```

- We can interpret and run our bad program
  - but we'll probably get a *runtime* error
- We'd like to get a *compile-time* error for such programs

Indexed state monad

```scala
// (Simple) State monad:
case class State[S, A](run: S ⇒ (S, A))

val simpleProg: State[Int, Unit] = for {
  num ← State.get // read an Int
  _ ← State.set(num + 1) // write an Int
} yield ()

// Indexed State monad:
case class IndexedState[F, T, A](run: F ⇒ (T, A))

val indexedProg: IndexedState[Int, String, Unit] = for {
  num ← IndexedState.get // read an Int
  _ ← IndexedState.set("str") // write a String!
} yield ()
```

Indexed state monad

```scala
// (Simple) State monad:
case class State[S, A](run: S ⇒ (S, A))

val simpleProg: State[Int, Unit] = for {
  num ← State.get // read an Int
  _ ← State.set(num + 1) // write an Int
} yield ()

// Indexed State monad:
case class IndexedState[F, T, A](run: F ⇒ (T, A))

val indexedProg: IndexedState[Int, String, Unit] = for {
  num ← IndexedState.get // read an Int
  _ ← IndexedState.set("str") // write a String!
} yield ()
```

Indexed state monad

```scala
// (Simple) State monad:
case class State[S, A](run: S ⇒ (S, A))

val simpleProg: State[Int, Unit] = for {
  num ← State.get // read an Int
  _ ← State.set(num + 1) // write an Int
} yield ()

// Indexed State monad:
case class IndexedState[F, T, A](run: F ⇒ (T, A))

val indexedProg: IndexedState[Int, String, Unit] = for {
  num ← IndexedState.get // read an Int
  _ ← IndexedState.set("str") // write a String!
} yield ()
```

Indexed state monad

```scala
// (Simple) State monad:
case class State[S, A](run: S ⇒ (S, A))

val simpleProg: State[Int, Unit] = for {
  num ← State.get // read an Int
  _ ← State.set(num + 1) // write an Int
} yield ()

// Indexed State monad:
case class IndexedState[F, T, A](run: F ⇒ (T, A))

val indexedProg: IndexedState[Int, String, Unit] = for {
  num ← IndexedState.get // read an Int
  _ ← IndexedState.set("str") // write a String!
} yield ()
```

Indexed state monad

```scala
// (Simple) State monad:
case class State[S, A](run: S ⇒ (S, A))

val simpleProg: State[Int, Unit] = for {
  num ← State.get // read an Int
  _ ← State.set(num + 1) // write an Int
} yield ()

// Indexed State monad:
case class IndexedState[F, T, A](run: F ⇒ (T, A))

val indexedProg: IndexedState[Int, String, Unit] = for {
  num ← IndexedState.get // read an Int
  _ ← IndexedState.set("str") // write a String!
} yield ()
```

Using IndexedState

```
class LoggedOut; class LoggedIn
trait UserApi {
  def login(c: Credentials):
    IndexedStateT[IO, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IndexedStateT[IO, LoggedIn, LoggedIn, String]
  def logout: IndexedStateT[IO, LoggedIn, LoggedOut, Unit]
}
val myProg: IndexedStateT[IO, LoggedOut, LoggedOut, String] =
  for {
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.runA(new LoggedOut)
```

Using IndexedState

```
class LoggedOut; class LoggedIn
trait UserApi {
  def login(c: Credentials):
    IndexedStateT[IO, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IndexedStateT[IO, LoggedIn, LoggedIn, String]
  def logout: IndexedStateT[IO, LoggedIn, LoggedOut, Unit]
}
val myProg: IndexedStateT[IO, LoggedOut, LoggedOut, String] =
  for {
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.runA(new LoggedOut)
```

Using IndexedState

```
class LoggedOut; class LoggedIn
trait UserApi {
  def login(c: Credentials):
    IndexedStateT[IO, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IndexedStateT[IO, LoggedIn, LoggedIn, String]
  def logout: IndexedStateT[IO, LoggedIn, LoggedOut, Unit]
}
val myProg: IndexedStateT[IO, LoggedOut, LoggedOut, String] =
  for {
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.runA(new LoggedOut)
```

Using IndexedState

```
class LoggedOut; class LoggedIn
trait UserApi {
  def login(c: Credentials):
    IndexedStateT[IO, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IndexedStateT[IO, LoggedIn, LoggedIn, String]
  def logout: IndexedStateT[IO, LoggedIn, LoggedOut, Unit]
}
val myProg: IndexedStateT[IO, LoggedOut, LoggedOut, String] =
  for {
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.runA(new LoggedOut)
```

Using IndexedState

```
class LoggedOut; class LoggedIn
trait UserApi {
  def login(c: Credentials):
    IndexedStateT[IO, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IndexedStateT[IO, LoggedIn, LoggedIn, String]
  def logout: IndexedStateT[IO, LoggedIn, LoggedOut, Unit]
}
val myProg: IndexedStateT[IO, LoggedOut, LoggedOut, String] =
  for {
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.runA(new LoggedOut)
```

Good and bad things

```
for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials) // compile error
} yield secret
```

The good: we get a *compile-time* error if we try to log in *after* reading the secret.

```
for {
  _ ← IndexedStateT.set[IO, LoggedOut, LoggedIn](
    new LoggedIn) // invalid operation!
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret
```

The bad: no compile-time error if we perform invalid state transitions.

Good and bad things

```
for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials) // compile error
} yield secret
```

The good: we get a *compile-time* error if we try to log in *after* reading the secret.

```
for {
  _ ← IndexedStateT.set[IO, LoggedOut, LoggedIn](
    new LoggedIn) // invalid operation!
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret
```

The bad: no compile-time error if we perform invalid state transitions.

Good and bad things

```
for {
  secret ← UserApi.readSecret
  // login AFTER readSecret
  _ ← UserApi.login(myCredentials) // compile error
} yield secret
```

The good: we get a *compile-time* error if we try to log in *after* reading the secret.

```
for {
  _ ← IndexedStateT.set[IO, LoggedOut, LoggedIn](
    new LoggedIn) // invalid operation!
  secret ← UserApi.readSecret
  _ ← UserApi.logout
} yield secret
```

The bad: no compile-time error if we perform invalid state transitions.

Further improvement: let's combine `IndexedStateT` with `Free`!

The *indexed* free monad

```
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

The *indexed* free monad

```
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

The *indexed* free monad

```scala
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

The *indexed* free monad

```scala
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

The *indexed* free monad

```
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

- Combines the advantages of both `IndexedState` and `Free`:

The *indexed* free monad

```scala
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

- Combines the advantages of both `IndexedState` and `Free`:
  - consistent *From* and *To* states

The *indexed* free monad

```
// (Simple) Free monad:
trait Free[S[_], A] {
  def flatMap[B](f: A ⇒ Free[S, B]): Free[S, B]
}

// Indexed Free monad:
trait IxFree[S[_, _, _], F, T, A] {
  def flatMap[B, U](f: A ⇒ IxFree[S, T, U, B]):
    IxFree[S, F, U, B]
}
```

- Combines the advantages of both `IndexedState` and `Free`:
  - consistent *From* and *To* states
  - we have control over the allowed operations

Using IxFree

```
trait UserApi {
  def login(c: Credentials):
    IxFree[UserOp, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IxFree[UserOp, LoggedIn, LoggedIn, String]
  def logout: IxFree[UserOp, LoggedIn, LoggedOut, Unit]
}
```

```
val myProg: IxFree[UserOp, LoggedOut, LoggedOut, String] =
  for {
    // we need to actually `login`:
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.foldMapA(interpreter)
```

Using IxFree

```scala
trait UserApi {
  def login(c: Credentials):
    IxFree[UserOp, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IxFree[UserOp, LoggedIn, LoggedIn, String]
  def logout: IxFree[UserOp, LoggedIn, LoggedOut, Unit]
}

val myProg: IxFree[UserOp, LoggedOut, LoggedOut, String] =
  for {
    // we need to actually `login`:
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.foldMapA(interpreter)
```

Using IxFree

```scala
trait UserApi {
  def login(c: Credentials):
    IxFree[UserOp, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IxFree[UserOp, LoggedIn, LoggedIn, String]
  def logout: IxFree[UserOp, LoggedIn, LoggedOut, Unit]
}

val myProg: IxFree[UserOp, LoggedOut, LoggedOut, String] =
  for {
    // we need to actually `login`:
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.foldMapA(interpreter)
```

Using IxFree

```
trait UserApi {
  def login(c: Credentials):
    IxFree[UserOp, LoggedOut, LoggedIn, Unit]
  def readSecret:
    IxFree[UserOp, LoggedIn, LoggedIn, String]
  def logout: IxFree[UserOp, LoggedIn, LoggedOut, Unit]
}

val myProg: IxFree[UserOp, LoggedOut, LoggedOut, String] =
  for {
    // we need to actually `login`:
    _ ← UserApi.login(myCredentials)
    secret ← UserApi.readSecret
    _ ← UserApi.logout
  } yield secret

val myIO: IO[String] =
  myProg.foldMapA(interpreter)
```

Summary

- Better (state machine-like) APIs:

Summary

- Better (state machine-like) APIs:
  - DSL-like operations

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:
    - disallow incorrect state transitions

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:
    - disallow incorrect state transitions
    - help with resource handling

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:
    - disallow incorrect state transitions
    - help with resource handling
  - When running the program:

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:
    - disallow incorrect state transitions
    - help with resource handling
  - When running the program:
    - disallow invalid start/end states

Summary

- Better (state machine-like) APIs:
  - DSL-like operations
  - compile-time errors for
    - incorrect state transitions
    - invalid operations
- We achieved this by:
  - using a free monad
  - which is *indexed* by the from/to states
- Further improvements:
  - When writing the interpreter:
    - disallow incorrect state transitions
    - help with resource handling
  - When running the program:
    - disallow invalid start/end states
  - These are implemented too, see the **source code** ...

Thank You!

`https://github.com/durban/scates`

- Source code, examples, slides, ...

Thank You!