Functional Programming in Scala

Typed Functional Programming in Scala

About Me

- Started software development ~10 years ago

About Me

- Started software development ~10 years ago
- I've begun with HTML, CSS, PHP, JavaScript

About Me

- Started software development ~10 years ago
- I've begun with HTML, CSS, PHP, JavaScript
- Early on I discovered LISP/Scheme and functional programming

About Me

- Started software development ~10 years ago
- I've begun with HTML, CSS, PHP, JavaScript
- Early on I discovered LISP/Scheme and functional programming
- Then Haskell and **typed** functional programming

About Me

- Started software development ~10 years ago
- I've begun with HTML, CSS, PHP, JavaScript
- Early on I discovered LISP/Scheme and functional programming
- Then Haskell and **typed** functional programming
- I wanted something similar to Haskell: FP + types

About Me

- Started software development ~10 years ago
- I've begun with HTML, CSS, PHP, JavaScript
- Early on I discovered LISP/Scheme and functional programming
- Then Haskell and **typed** functional programming
- I wanted something similar to Haskell: FP + types
- Switched to Scala ~5 years ago

About Me

- Today, contractor for eloquentix

About Me

- Today, contractor for eloquentix
- Current project is about controlling power plant assets

About Me

- Today, contractor for eloquentix
- Current project is about controlling power plant assets
- They output electricity in the German and UK national grids

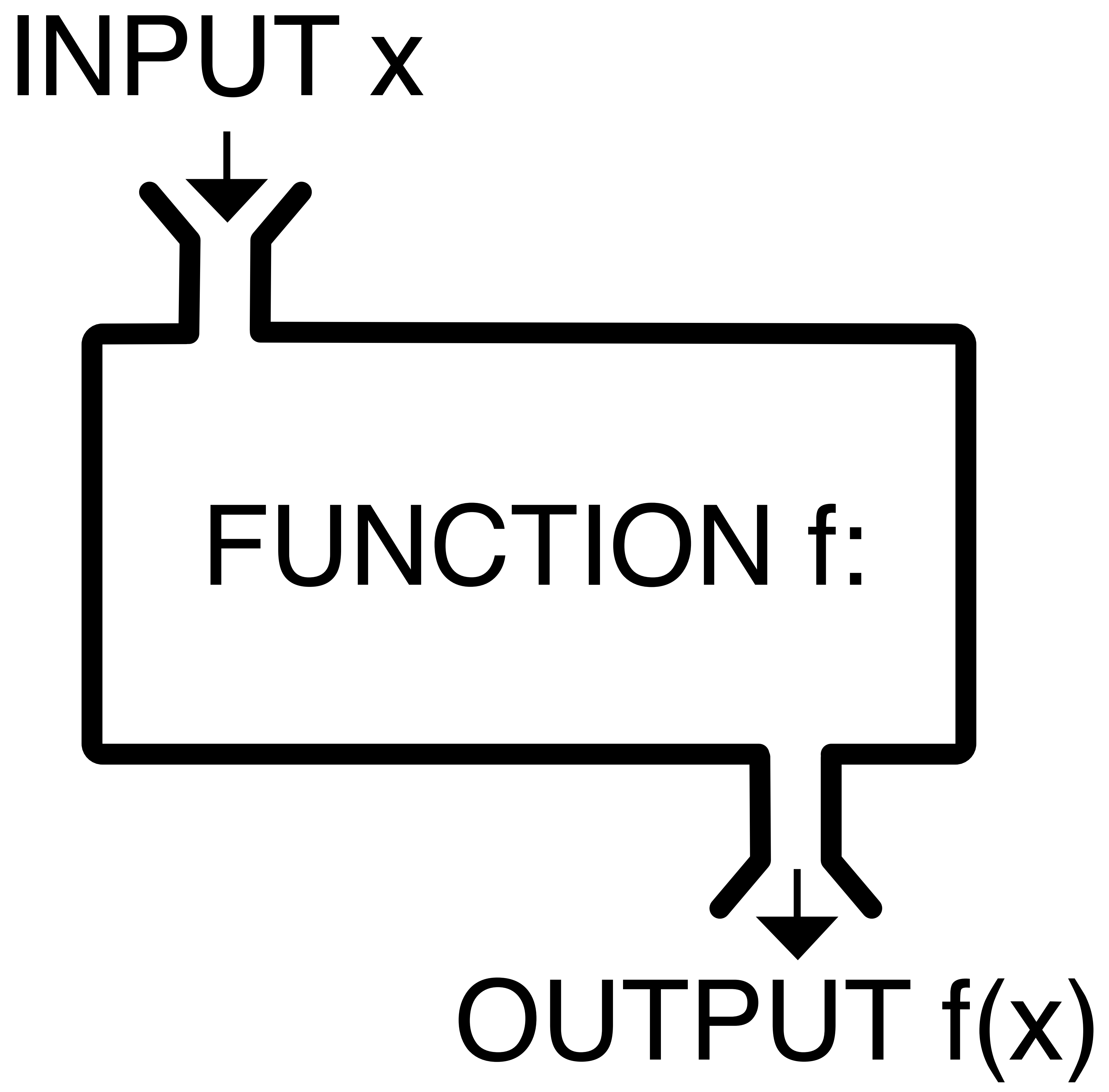
About Me

- Today, contractor for eloquentix
- Current project is about controlling power plant assets
- They output electricity in the German and UK national grids
- I'm quite excited about this project :)

Today's Plan

- Programming with functions and values
- Taking advantage of types
- Finite state machines

Functional Programming


INPUT x
FUNCTION f:
OUTPUT f(x)

Functional Programming

- Promming with functions, not procedures

Functional Programming

- Promming with functions, not procedures
- A function's output depends entirely on its input

Functional Programming

- Promming with functions, not procedures
- A function's output depends entirely on its input
- Everything it does, it does based on the arguments it takes

Functional Programming

- Promming with functions, not procedures
- A function's output depends entirely on its input
- Everything it does, it does based on the arguments it takes
- What it does is reflected in the return value

Functional Programming

- Promming with functions, not procedures
- A function's output depends entirely on its input
- Everything it does, it does based on the arguments it takes
- What it does is reflected in the return value
- Everything else is a **side-effect**

Functional Programming

- Promming with functions, not procedures
- A function's output depends entirely on its input
- Everything it does, it does based on the arguments it takes
- What it does is reflected in the return value
- Everything else is a **side-effect**
- Side-effects are the purpose of any app, but hard to reason about

Functional Programming

- When we write programs, we reason by substitution

Functional Programming

- When we write programs, we reason by substitution
- This is called equational reasoning — we replace equals for equals, **syntactically**

Functional Programming

- When we write programs, we reason by substitution
- This is called equational reasoning — we replace equals for equals, **syntactically**
- This is called the substitution model; we compute by substitution

Equational Reasoning #1

```
val a = 1
val b = a + a
val c = 2 * b
val b = 1 + 1
val c = 2 * b
val b = 2
val c = 2 * b
val c = 2 * 2
val c = 4
```

Equational Reasoning #2

```
def length[A](list: List[A]): Int =
```

```scala
def length[A](list: List[A]): Int =
  list match {


  }
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil =>

  }
```

```
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0

  }
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail =>
  }
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
1 + length(List(2, 3))
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
1 + length(List(2, 3))
1 + 1 + length(List(3))
```

```
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
1 + length(List(2, 3))
1 + 1 + length(List(3))
1 + 1 + 1 + length(List())
```

```scala
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
1 + length(List(2, 3))
1 + 1 + length(List(3))
1 + 1 + 1 + length(List())
1 + 1 + 1 + 0
```

```
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case _ :: tail => 1 + length(tail)
  }

length(List(1, 2, 3))
1 + length(List(2, 3))
1 + 1 + length(List(3))
1 + 1 + 1 + length(List())
1 + 1 + 1 + 0
3
```

Imperative Reasoning

```
var a = 1
a = a + a
a = 2 * a
```

```
1 = 1 + 1
1 = 2 * 1
```

```
var a = 1
a = a + a
a = 2 * a
```

```
val a₀ = 1
val a₁ = a₀ + a₀
val a₂ = 2 * a₁
```

Time-Varying Values

```
val a₀ = 1
val a₁ = a₀ + a₀
val a₂ = 2 * a₁
```

Even compilers do this to ease optimizations/program transformations:
https://en.m.wikipedia.org/wiki/Static_single_assignment_form

Equational Reasoning #3

```scala
import scala.io.StdIn

object ConsoleCalculator {
  def main(args: Array[String]): Unit = {
    println("Enter number: ")
    val a = StdIn.readInt()

    println("Enter number: ")
    val b = StdIn.readInt()

    println(s"a + b = ${a + b}")
  }
}
```

```scala
import scala.io.StdIn

object ConsoleCalculator {
  def main(args: Array[String]): Unit = {
    val a = number
    val b = number

    println(s"a + b = ${a + b}")
  }

  private val number: Int = {
    println("Enter number: ")
    StdIn.readInt()
  }
}
```

```scala
import scala.io.StdIn

object ConsoleCalculator {
  def main(args: Array[String]): Unit = {
    val a = number()
    val b = number()

    println(s"a + b = ${a + b}")
  }

  // Delay effects by using a procedure.
  // This is not a function, as it takes no params.
  private def number(): Int = {
    println("Enter number: ")
    StdIn.readInt()
  }
}
```

Delaying Effects

- We can preserve the meaning of the program by manually deferring all side-effects in our code, just like in the previous example

Delaying Effects

- We can preserve the meaning of the program by manually deferring all side-effects in our code, just like in the previous example
- This is exactly what functional programming does

Delaying Effects

- We can preserve the meaning of the program by manually deferring all side-effects in our code, just like in the previous example
- This is exactly what functional programming does
- It always starts with values, though, not just when necessary

Delaying Effects

- We can preserve the meaning of the program by manually deferring all side-effects in our code, just like in the previous example
- This is exactly what functional programming does
- It always starts with values, though, not just when necessary
- In functional programming, side-effects are represented as values by delaying them.

Benefits

- Easier to reason about, i.e., we can perform substition mentally

Benefits

- Easier to reason about, i.e., we can perform substition mentally
- Eliminates the time variable from reasoning

Benefits

- Easier to reason about, i.e., we can perform substition mentally
- Eliminates the time variable from reasoning
- Easier to refactor code, e.g., extract or inline variable/method

Benefits

- Easier to reason about, i.e., we can perform substition mentally
- Eliminates the time variable from reasoning
- Easier to refactor code, e.g., extract or inline variable/method
- Easier to glue together pieces of code; each piece depends only on arguments, not context in which it's used

Benefits

- Easier to reason about, i.e., we can perform substition mentally
- Eliminates the time variable from reasoning
- Easier to refactor code, e.g., extract or inline variable/method
- Easier to glue together pieces of code; each piece depends only on arguments, not context in which it's used
- The last point leads to composition

Benefits

- Easier to reason about, i.e., we can perform substition mentally
- Eliminates the time variable from reasoning
- Easier to refactor code, e.g., extract or inline variable/method
- Easier to glue together pieces of code; each piece depends only on arguments, not context in which it's used
- The last point leads to composition
- Composition: build larger programs out of smaller ones

Typed Functional Programming

Typed Functional Programming

- Scala is a typed language; use this to your advantage

Typed Functional Programming

- Scala is a typed language; use this to your advantage
- Encode business logic in types

Typed Functional Programming

- Scala is a typed language; use this to your advantage
- Encode business logic in types
- Useful when business rules change

Typed Functional Programming

- Scala is a typed language; use this to your advantage
- Encode business logic in types
- Useful when business rules change
- Useful as pseudo-documentation (but insufficient)

Typed Functional Programming

- Scala is a typed language; use this to your advantage
- Encode business logic in types
- Useful when business rules change
- Useful as pseudo-documentation (but insufficient)
- Type system: a companion that watches your back for stupid mistakes

Typed Functional Programming

- Scala is a typed language; use this to your advantage
- Encode business logic in types
- Useful when business rules change
- Useful as pseudo-documentation (but insufficient)
- Type system: a companion that watches your back for stupid mistakes
- Type system: a companion that guides your implementation

Type Safety — Exhibit A

```scala
val text = "Hi!"
val html = "<h1>Hi!</h1>"

sendEmail("ionut.g.stan@gmail.com", html, text)
```

```scala
def sendEmail(to: String, text: String, html: String): Unit = ???

val text = "Hi!"
val html = "<h1>Hi!</h1>"

sendEmail("ionut.g.stan@gmail.com", html, text)
```

"Make illegal states unrepresentable."

https://vimeo.com/14313378

```scala
def sendEmail(to: String, text: String, html: String): Unit = ???

val text = "Hi!"
val html = "<h1>Hi!</h1>"

sendEmail("ionut.g.stan@gmail.com", html, text)
```

```scala
case class Text(value: String)
case class HTML(value: String)

def sendEmail(to: String, text: String, html: String): Unit = ???

val text = "Hi!"
val html = "<h1>Hi!</h1>"

sendEmail("ionut.g.stan@gmail.com", html, text)
```

```scala
case class Text(value: String)
case class HTML(value: String)

def sendEmail(to: String, text: Text, html: HTML): Unit = ???

val text = "Hi!"
val html = "<h1>Hi!</h1>"

sendEmail("ionut.g.stan@gmail.com", html, text)
```

```scala
case class Text(value: String)
case class HTML(value: String)

def sendEmail(to: String, text: Text, html: HTML): Unit = ???

val text = Text("Hi!")
val html = HTML("<h1>Hi!</h1>")

sendEmail("ionut.g.stan@gmail.com", html, text)
```

```scala
case class Text(value: String)
case class HTML(value: String)

def sendEmail(to: String, text: Text, html: HTML): Unit = ???

val text = Text("Hi!")
val html = HTML("<h1>Hi!</h1>")

sendEmail("ionut.g.stan@gmail.com", text, html)
```

```scala
case class Text(value: String) extends AnyVal
case class HTML(value: String) extends AnyVal

def sendEmail(to: String, text: Text, html: HTML): Unit = ???

val text = Text("Hi!")
val html = HTML("<h1>Hi!</h1>")

sendEmail("ionut.g.stan@gmail.com", text, html)
```

Type Safety — Exhibit B

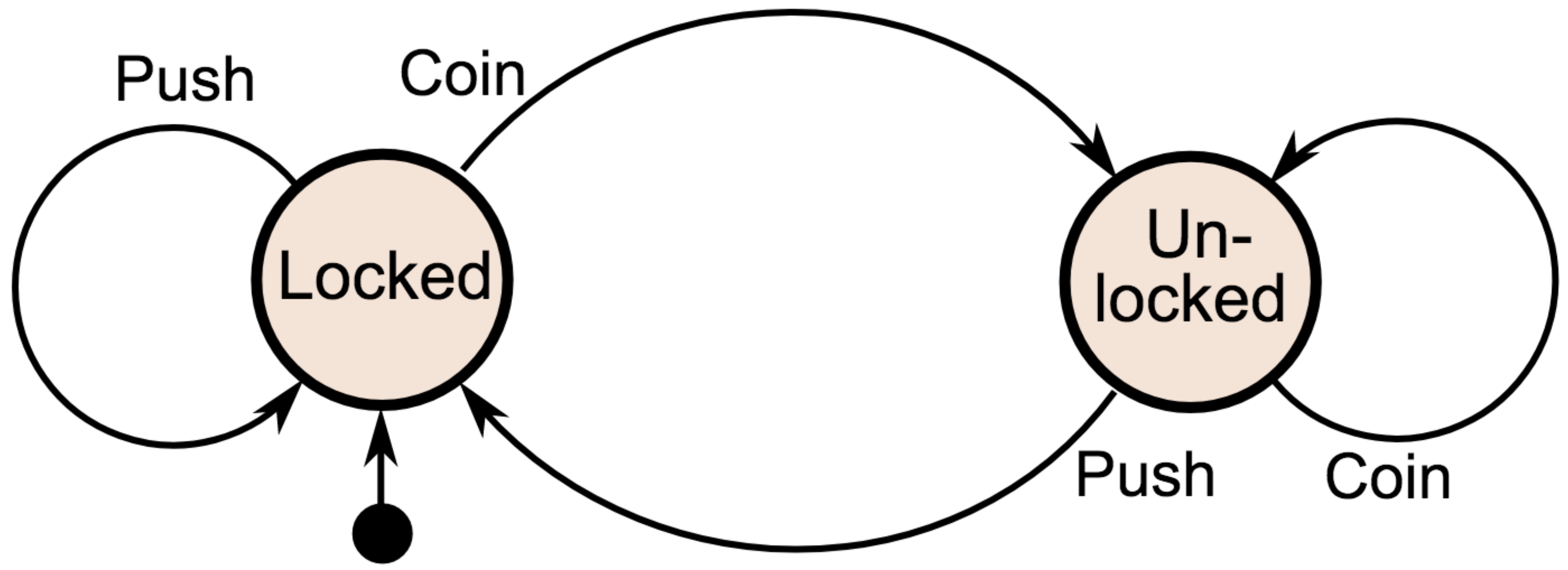
```scala
def user(id: Long): Future[User] =
  Users.all.filter(_.id === id)
```

```scala
def user(id: User.ID): Future[User] =
  Users.all.filter(_.id === id)
```

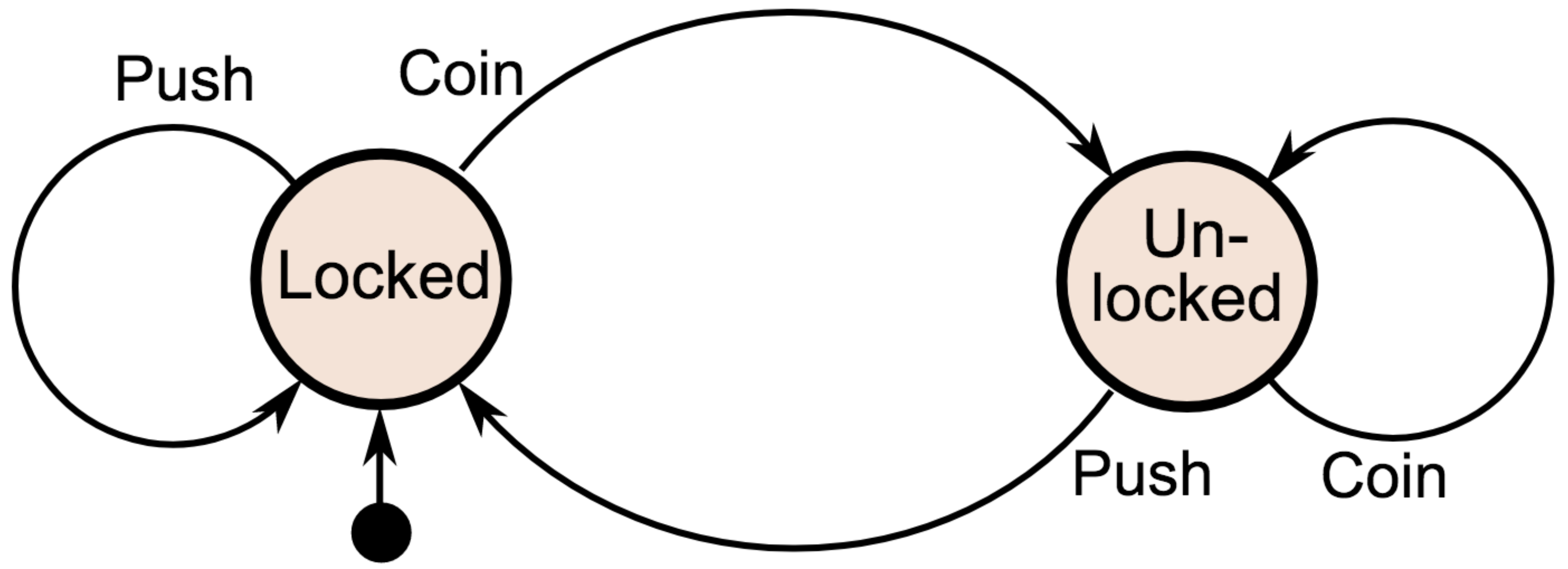
```scala
case class User(id: User.ID, email: String)

object User {
  case class ID(value: Long) extends AnyVal
}

def user(id: User.ID): Future[User] =
  Users.all.filter(_.id === id) // Slick code
```

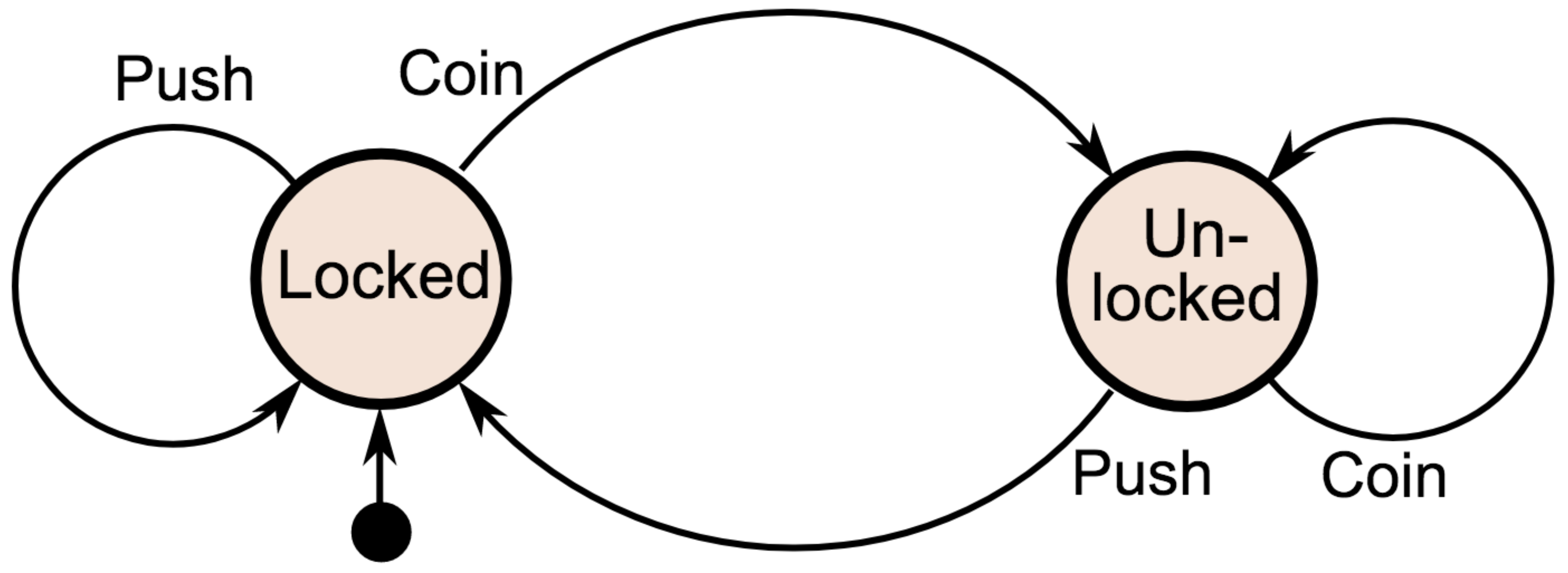
Type Safety — Exhibit C

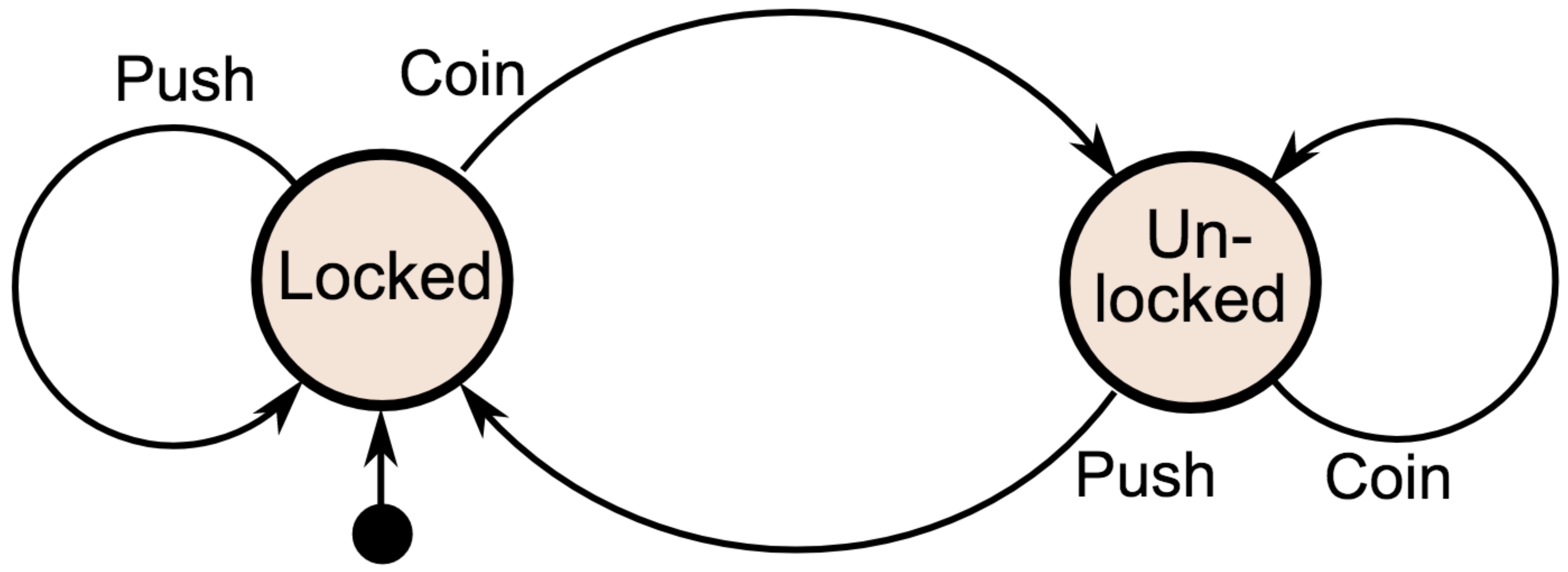
Finite State Machines


Push
Coin
Locked
Un-
locked
Push
Coin

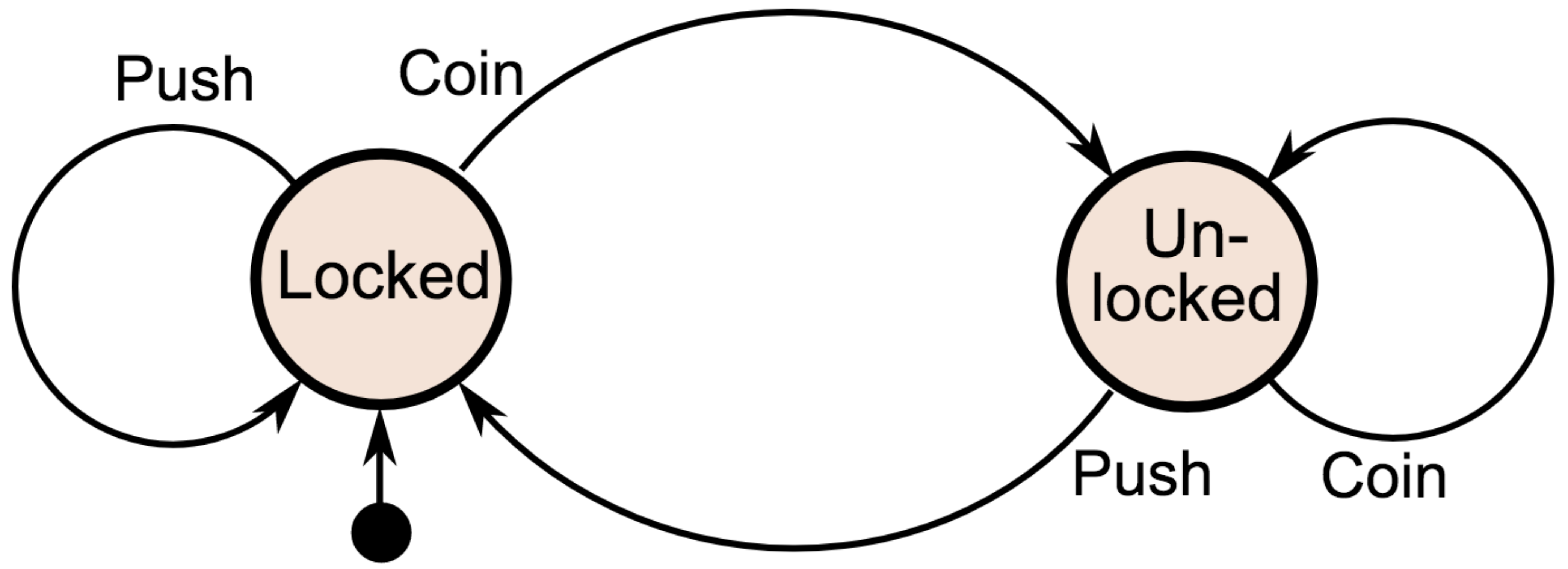
Action\State	Locked	Unlocked
Coin	Unlocked	Unlocked
Push	Locked	Locked


Push
Coin
Locked
Un-
locked
Push
Coin

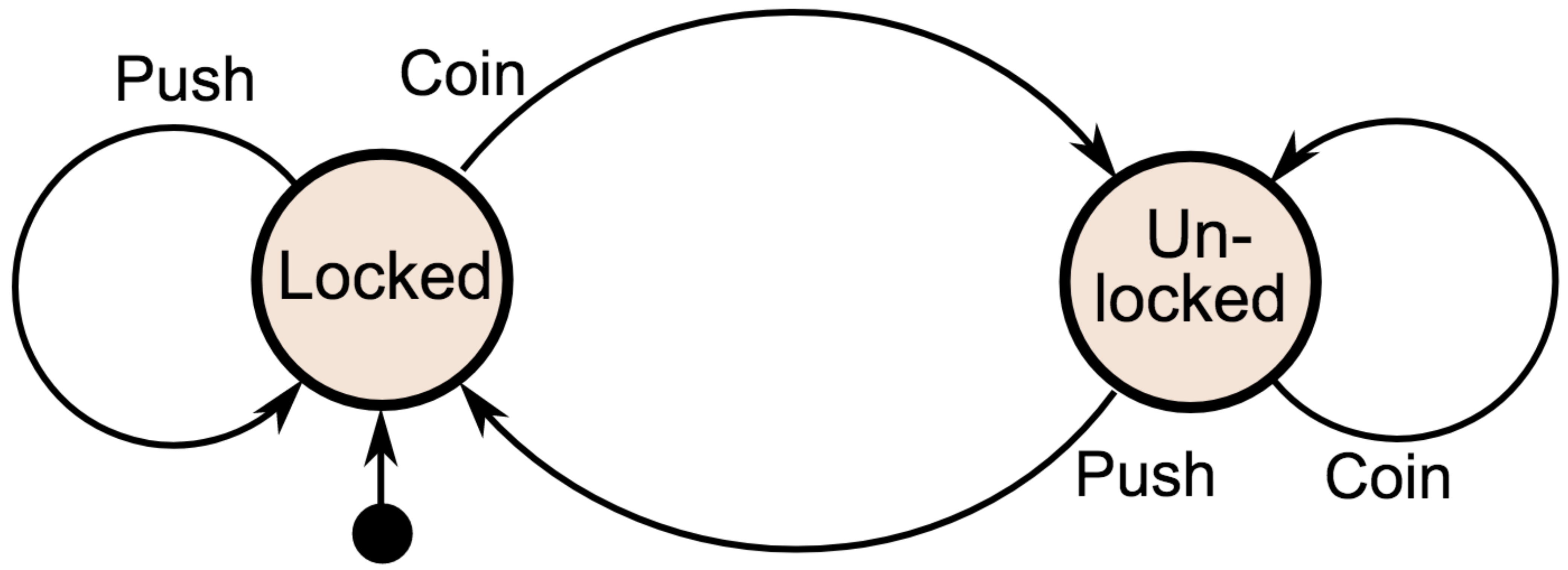

Push
Coin
Locked
Un-
locked
Push
Coin



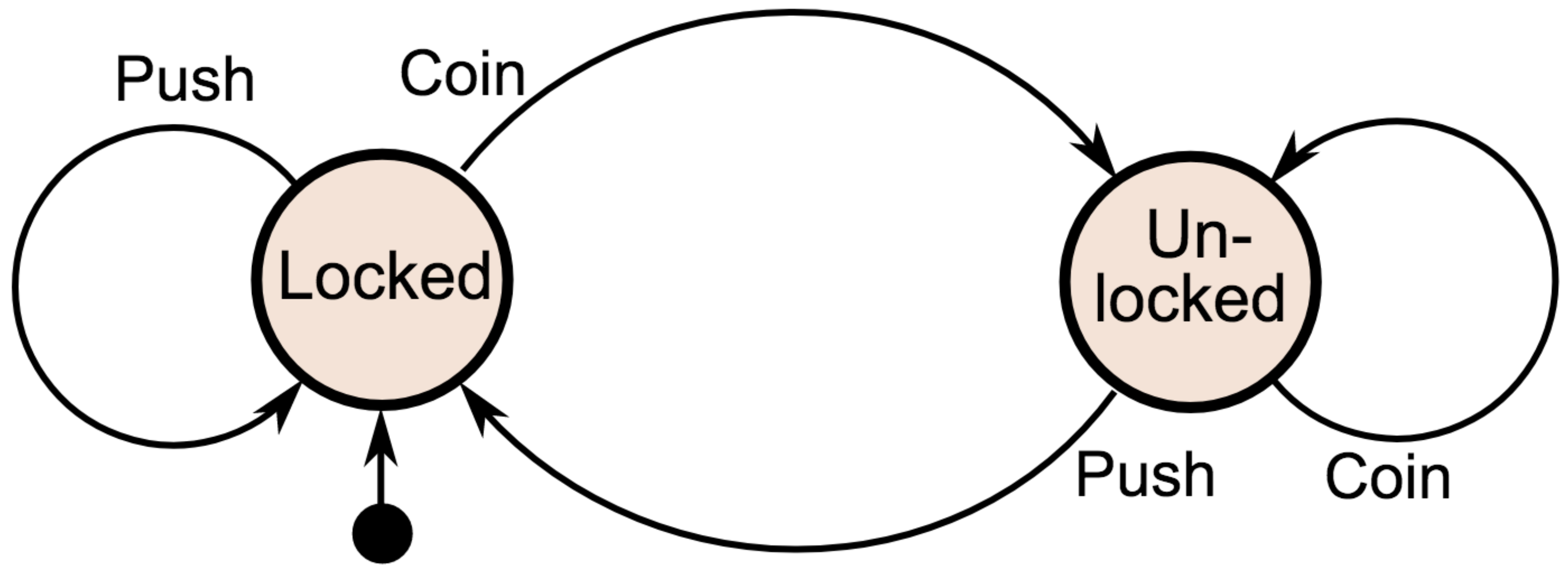
```
trait Locked {


}
```



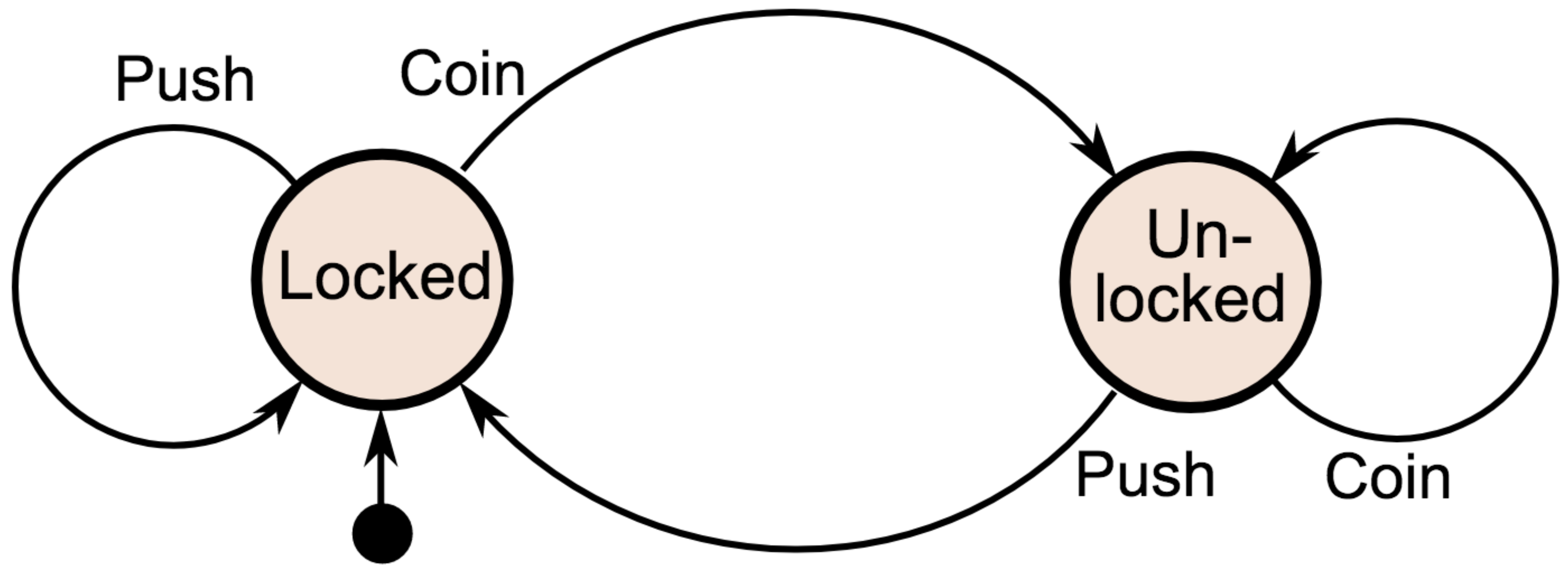
```
trait Locked {


}

trait Unlocked {


}
```



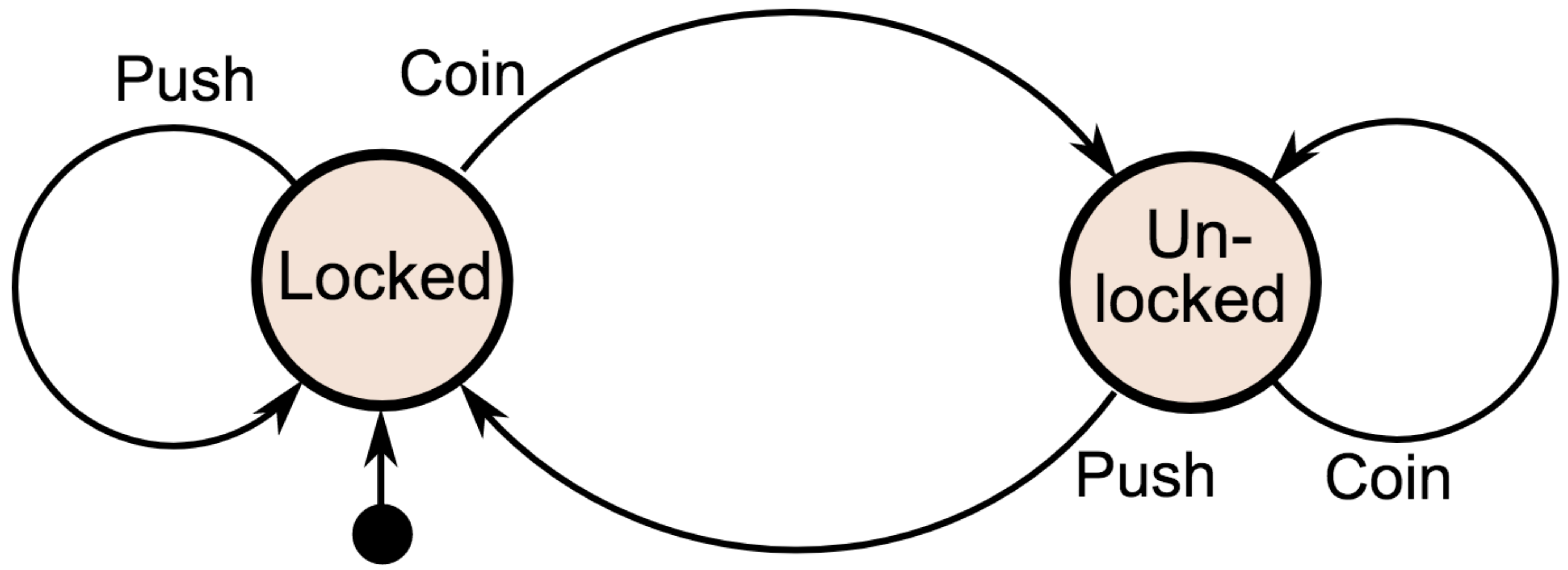
```
trait Locked {
  def push: Locked

}

trait Unlocked {



}
```



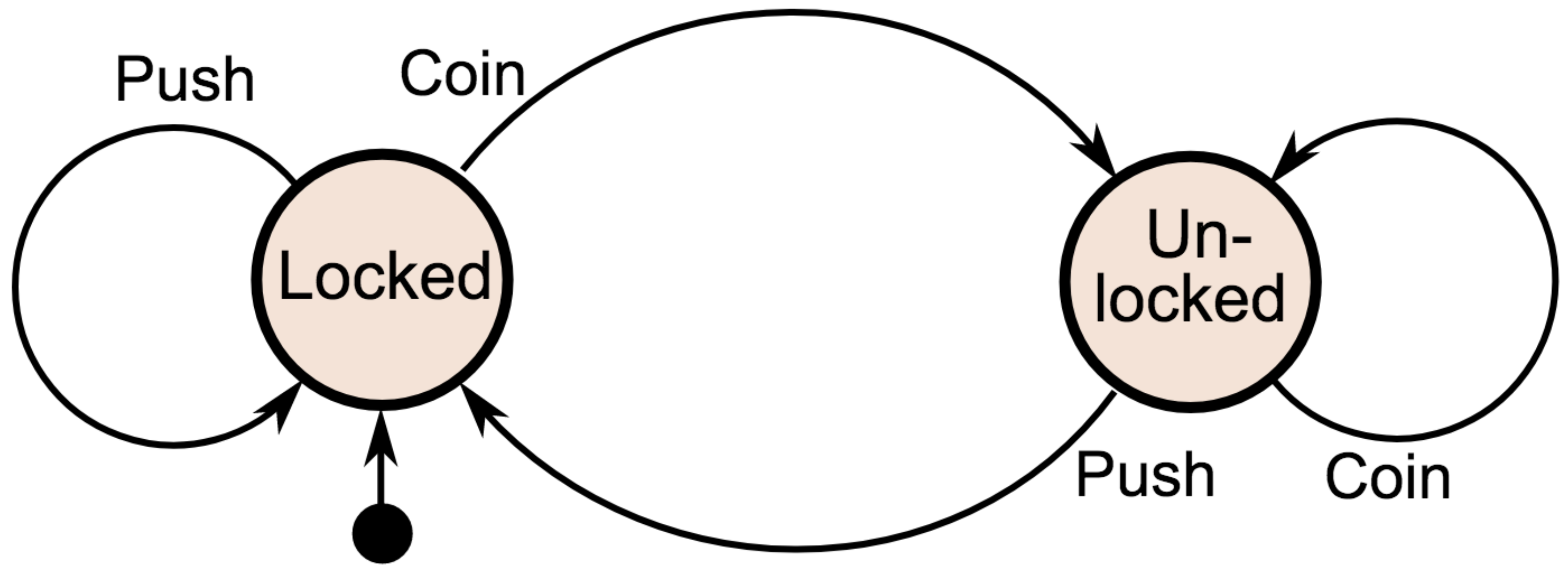
```
trait Locked {
  def push: Locked
  def coin: Unlocked
}

trait Unlocked {



}
```



```
trait Locked {
  def push: Locked
  def coin: Unlocked
}

trait Unlocked {
  def push: Locked


}
```



```
trait Locked {
  def push: Locked
  def coin: Unlocked
}

trait Unlocked {
  def push: Locked
  def coin: Unlocked
}
```



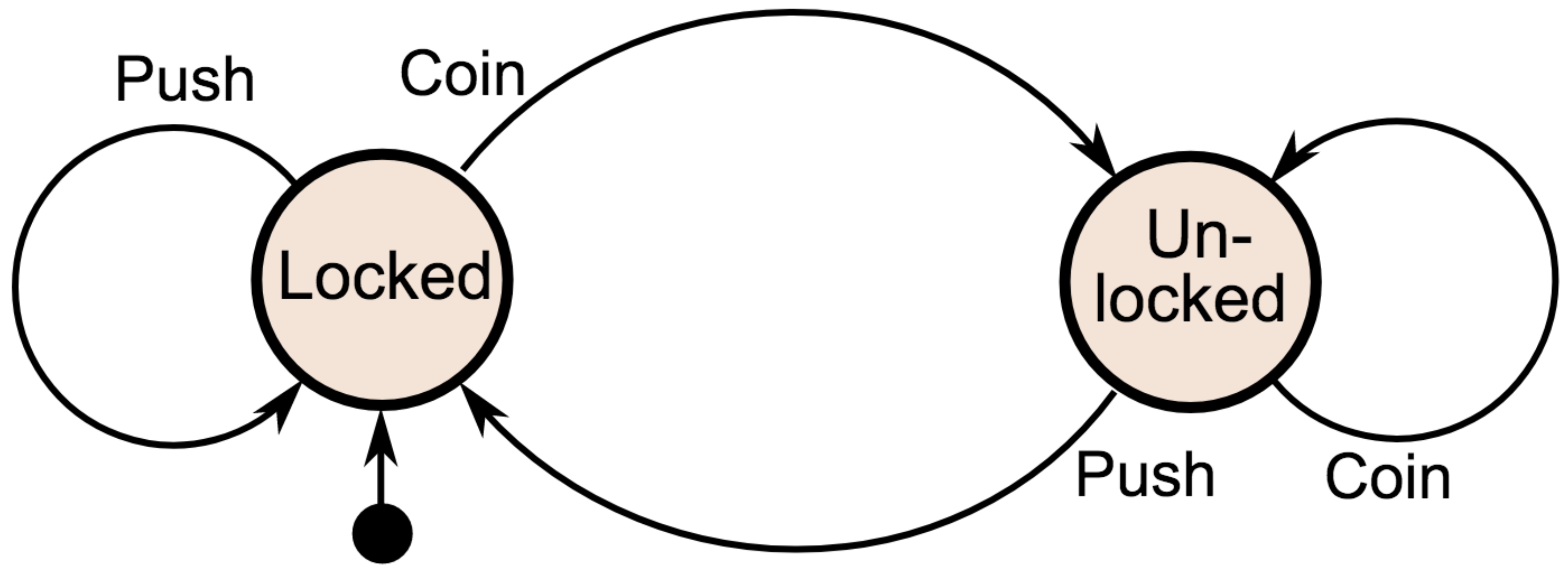
```
trait Locked {
  ~~def push: Locked~~
  def coin: Unlocked
}

trait Unlocked {
  def push: Locked
  ~~def coin: Unlocked~~
}
```

Phantom Types

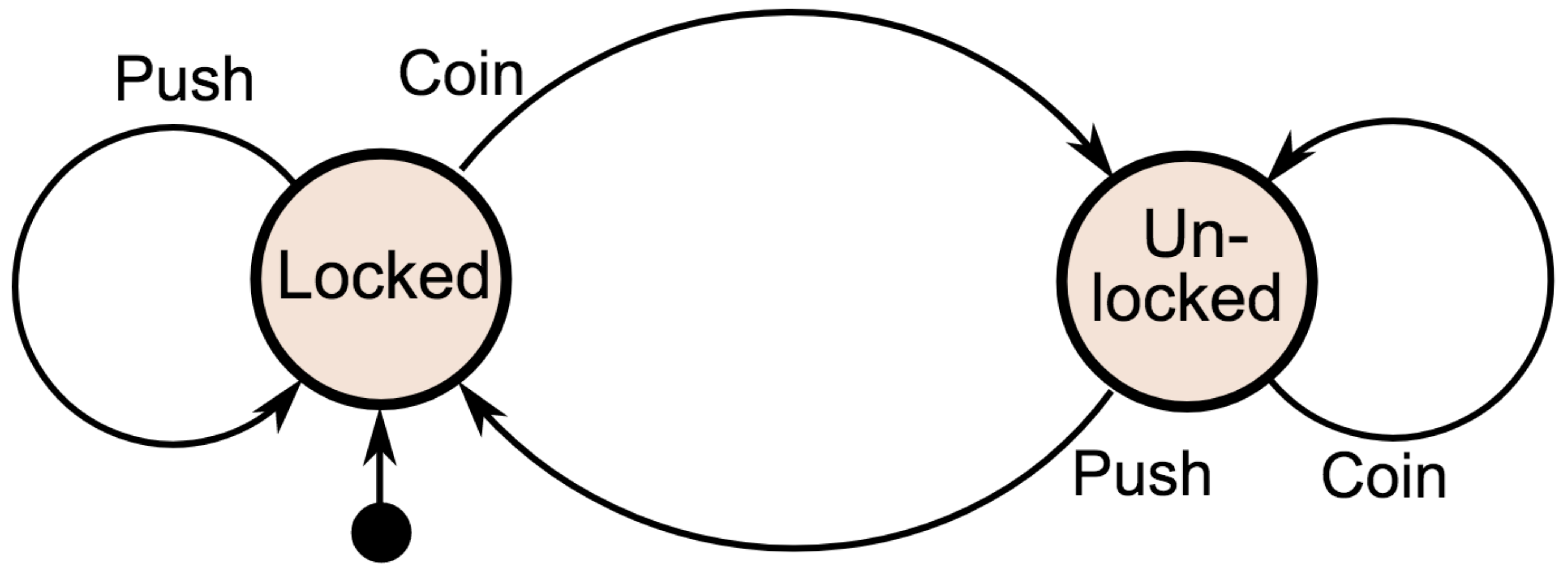
```scala
sealed trait State

object State {
  sealed trait Locked extends State
  sealed trait Unlocked extends State
}

trait Turnstile[S <: State] {
  def push: Turnstile[State.Locked]
  def coin: Turnstile[State.Unlocked]
}
```

```scala
sealed trait State

object State {
  sealed trait Locked extends State
  sealed trait Unlocked extends State
}

trait Turnstile[S <: State] {
  def push(implicit evidence: S =:= State.Unlocked): Turnstile[State.Locked]
  def coin(implicit evidence: S =:= State.Locked): Turnstile[State.Unlocked]
}
```

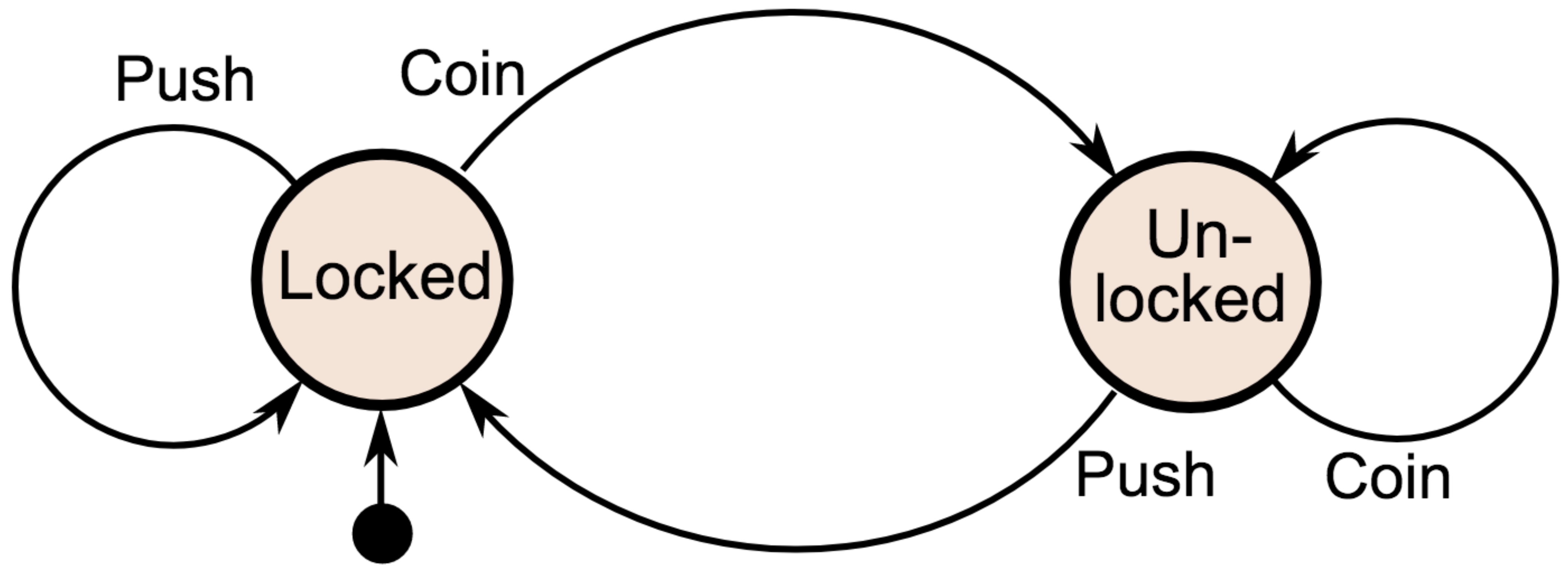
Finite State Machimes + Actors



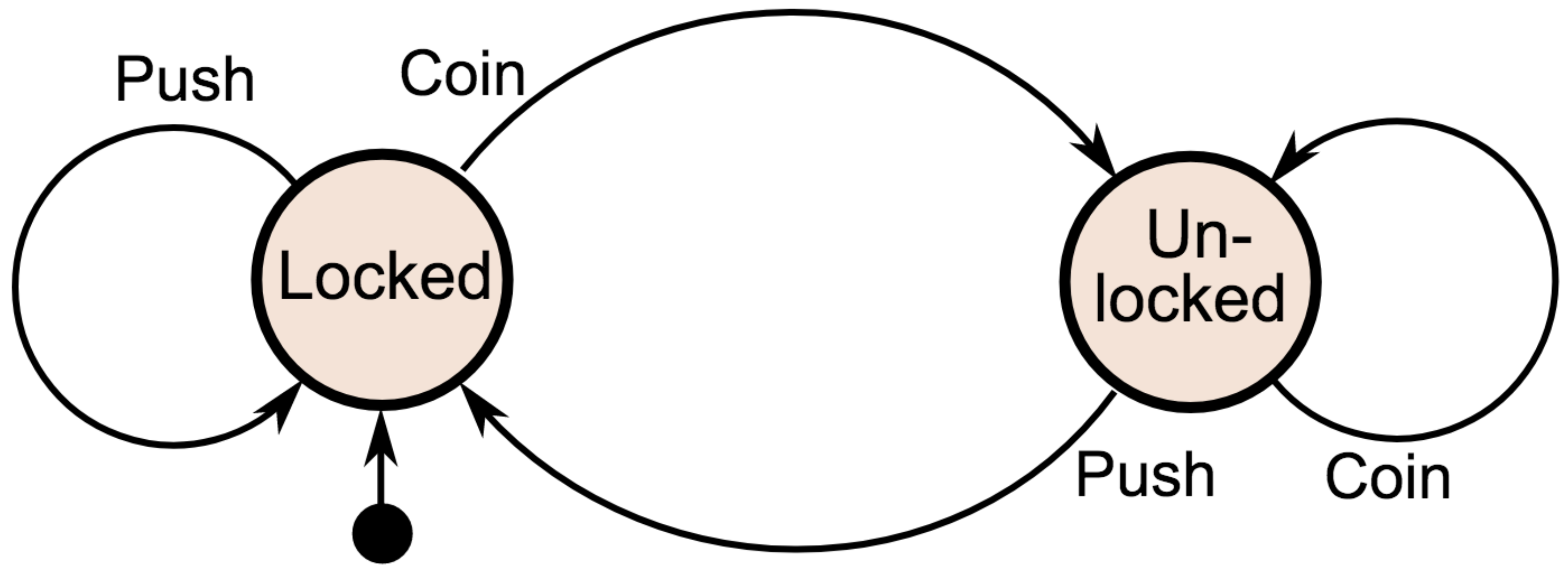
```
final class TurnstileActor extends Actor {








}
```



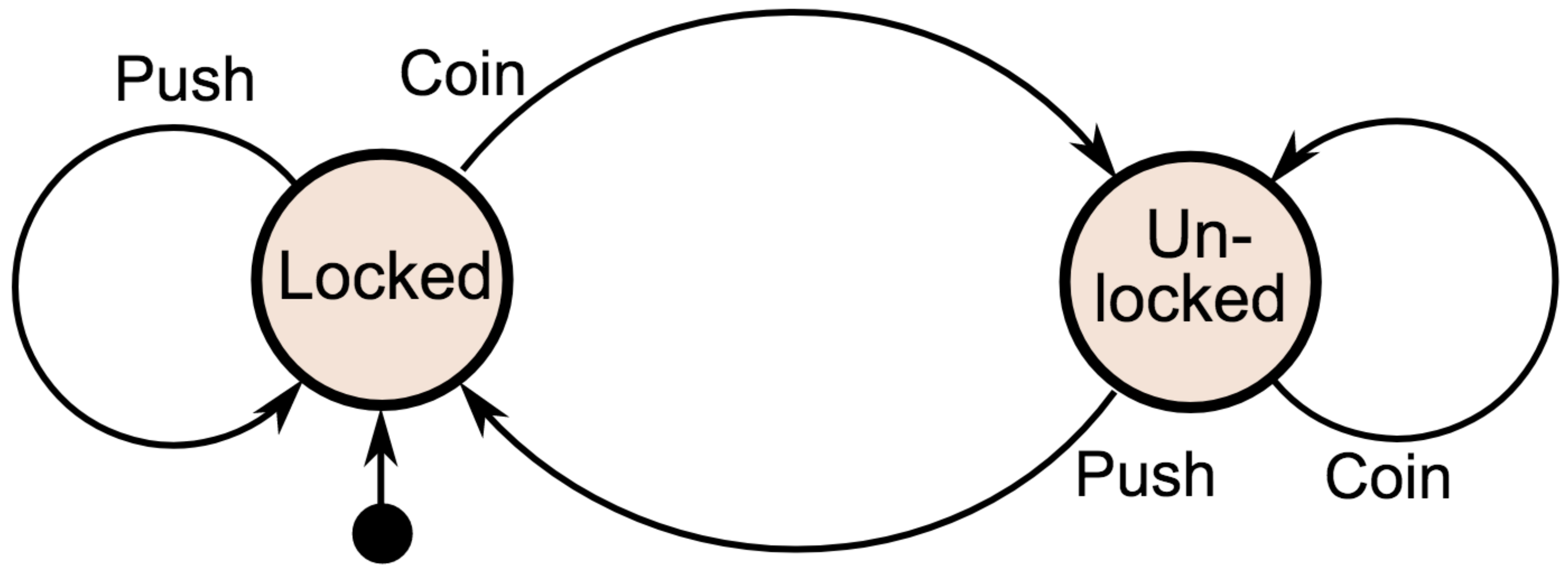
```
final class TurnstileActor extends Actor {



  def locked: Receive = {


  }





}
```



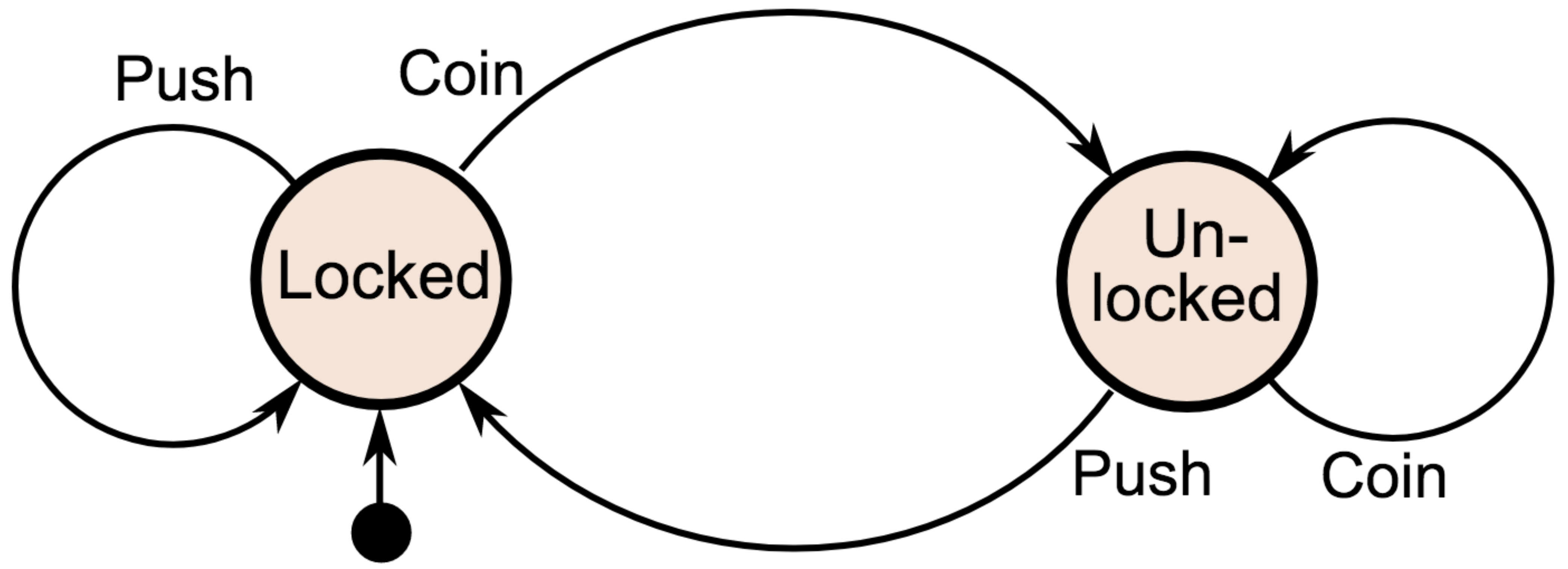
```
final class TurnstileActor extends Actor {



  def locked: Receive = {


  }

  def unlocked: Receive = {


  }
}
```



```scala
final class TurnstileActor extends Actor {

  def receive: Receive = locked

  def locked: Receive = {


  }

  def unlocked: Receive = {


  }
}
```



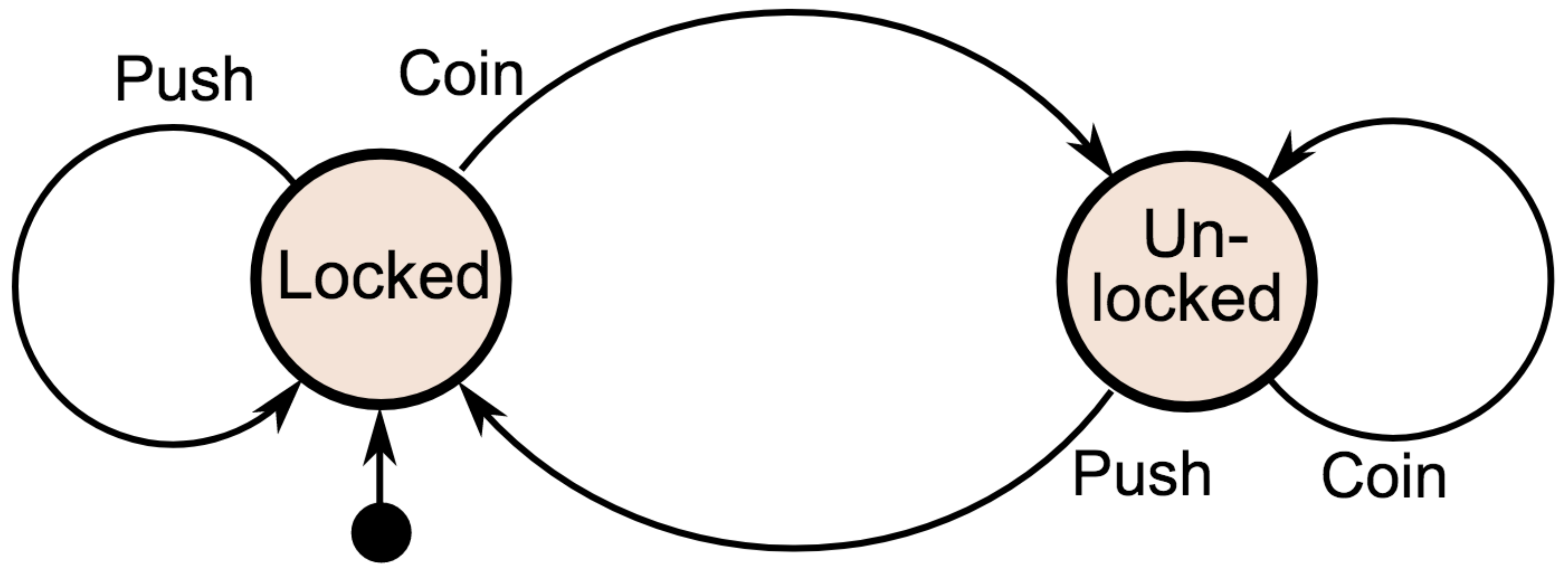
```
final class TurnstileActor extends Actor {


  def receive: Receive = locked

  def locked: Receive = {


  }

  def unlocked: Receive = {



  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



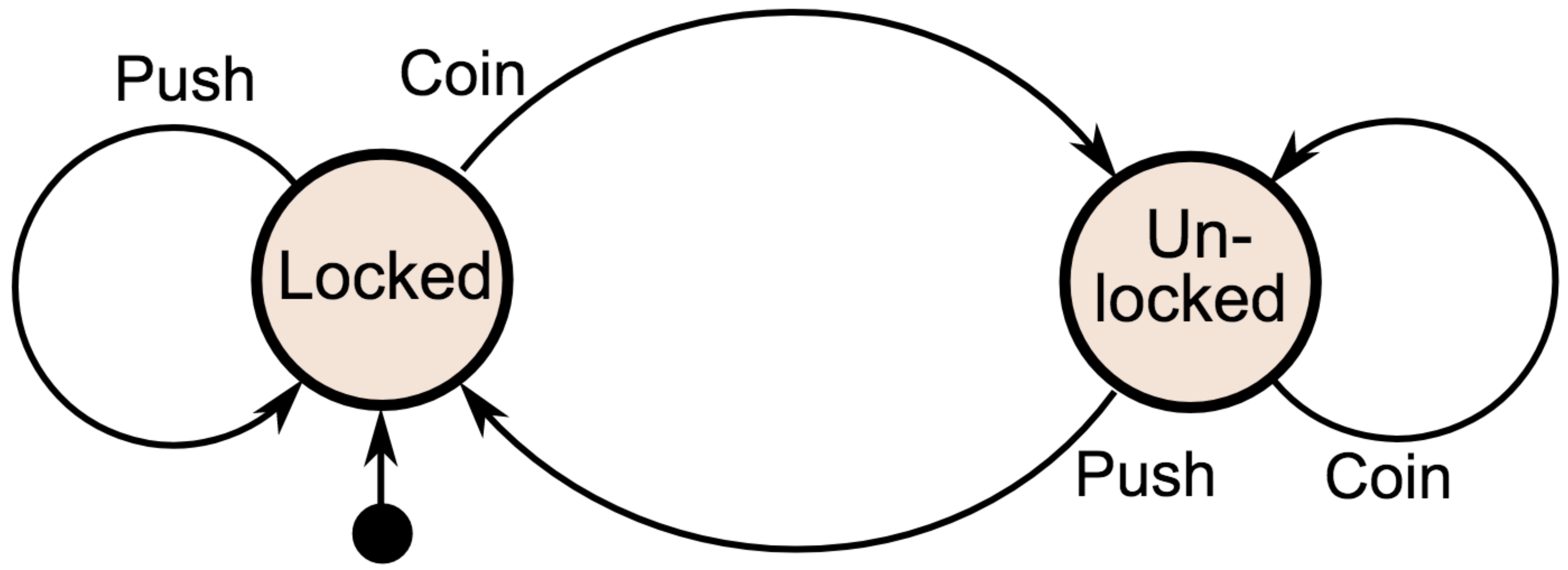
```
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin =>
    case Push =>
  }

  def unlocked: Receive = {
    case Coin =>
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```

Action\State	Locked	Unlocked
Coin	Unlocked	Unlocked
Push	Locked	Locked

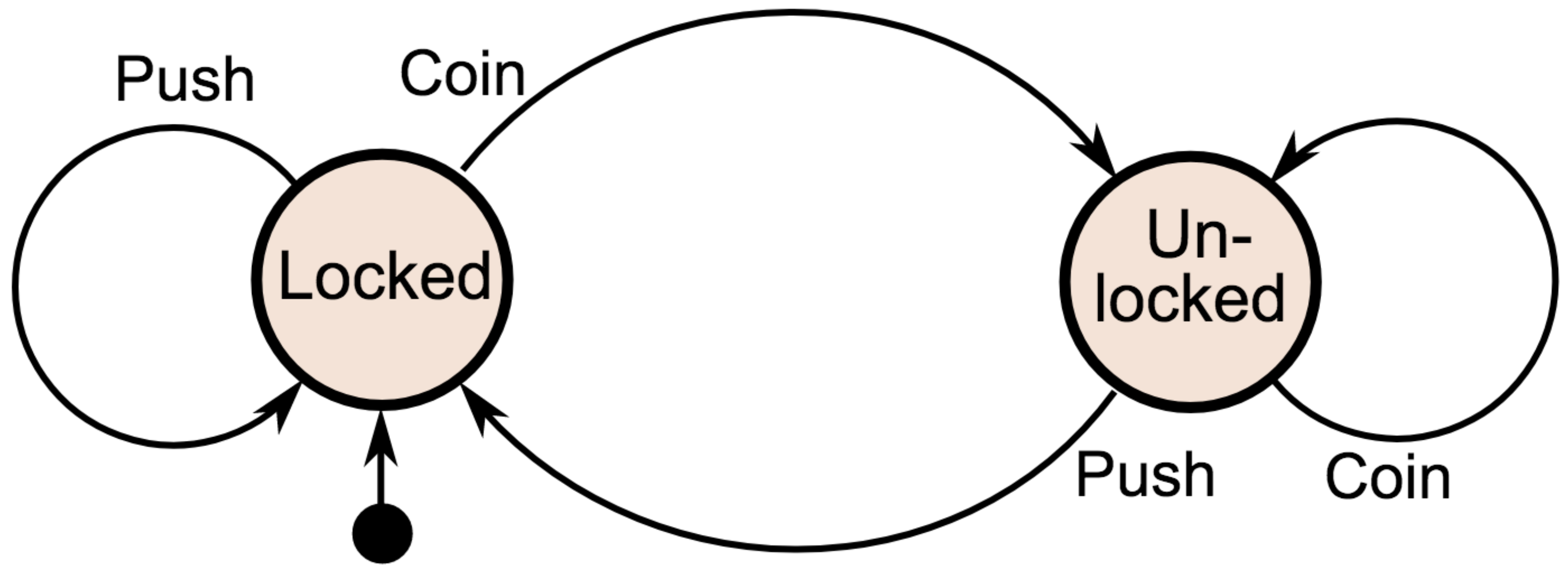
```
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin =>
    case Push =>
  }

  def unlocked: Receive = {
    case Coin =>
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



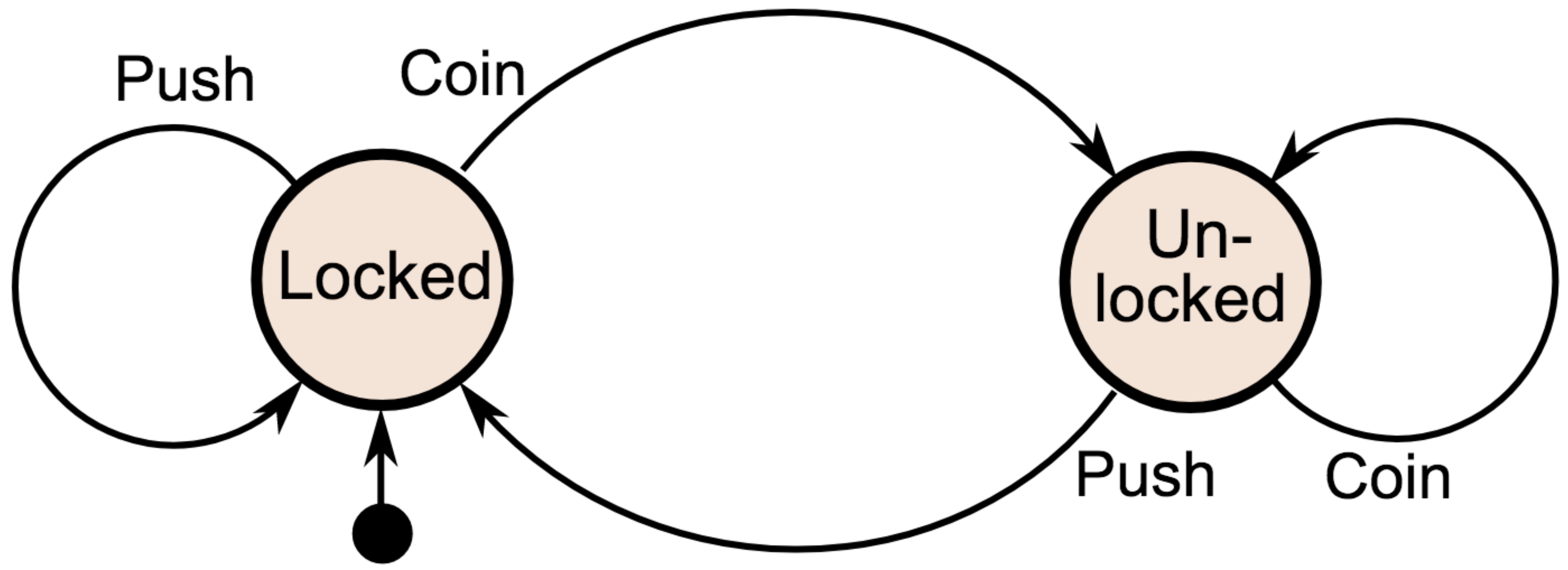
```
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin =>
    case Push =>
  }

  def unlocked: Receive = {
    case Coin =>
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



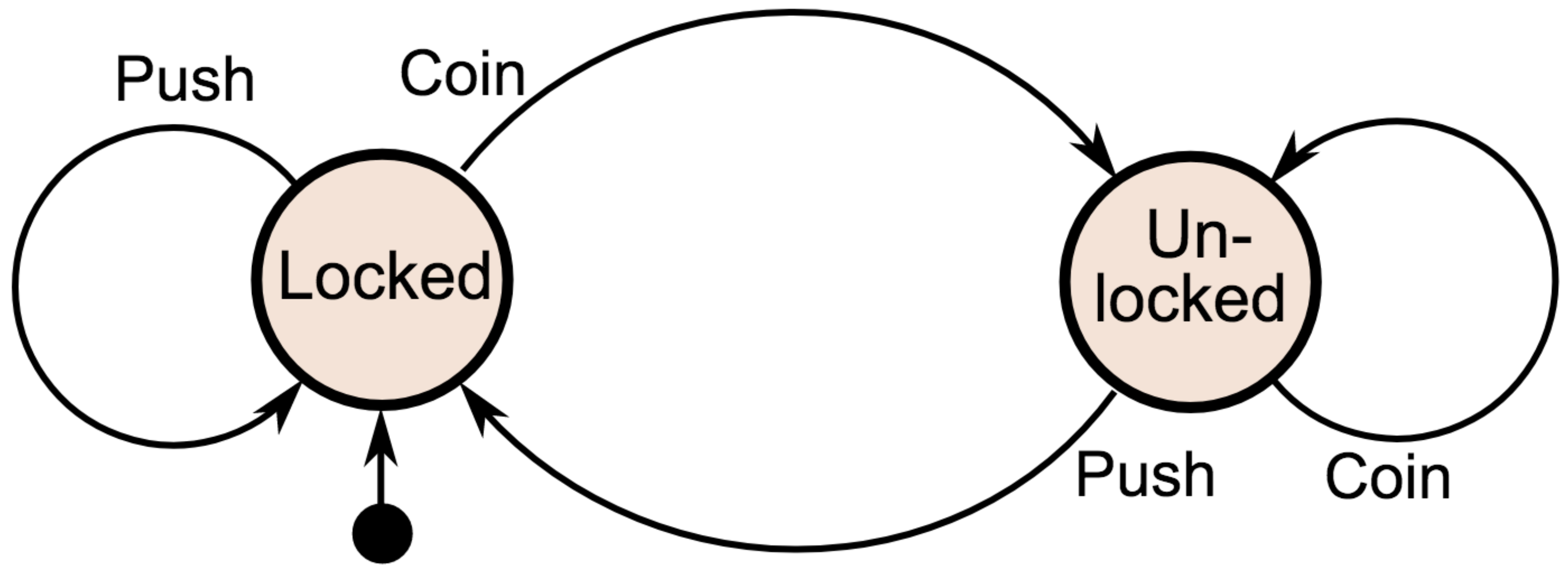
```
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin => context.become(unlocked)
    case Push =>
  }

  def unlocked: Receive = {
    case Coin =>
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



```
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin => context.become(unlocked)
    case Push => () // emit warning, maybe
  }

  def unlocked: Receive = {
    case Coin =>
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



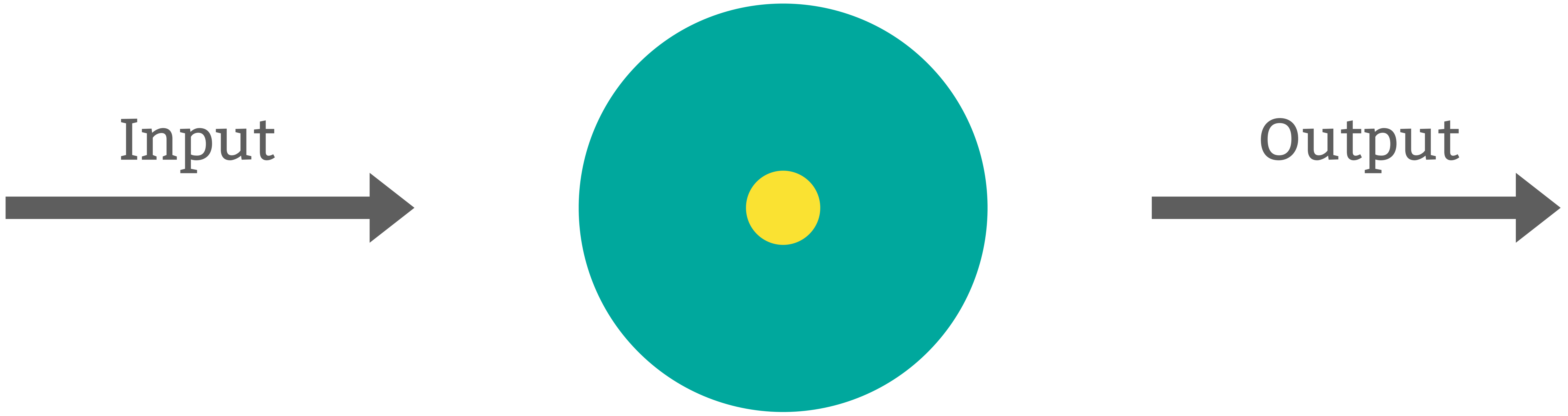
```scala
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin => context.become(unlocked)
    case Push => () // emit warning, maybe
  }

  def unlocked: Receive = {
    case Coin => () // refuse coin, maybe
    case Push =>
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```



```scala
final class TurnstileActor extends Actor {
  import TurnstileActor._

  def receive: Receive = locked

  def locked: Receive = {
    case Coin => context.become(unlocked)
    case Push => () // emit warning, maybe
  }

  def unlocked: Receive = {
    case Coin => () // refuse coin, maybe
    case Push => context.become(locked)
  }
}

object TurnstileActor {
  case object Coin
  case object Push
}
```

Functional Core, Imperative Shell

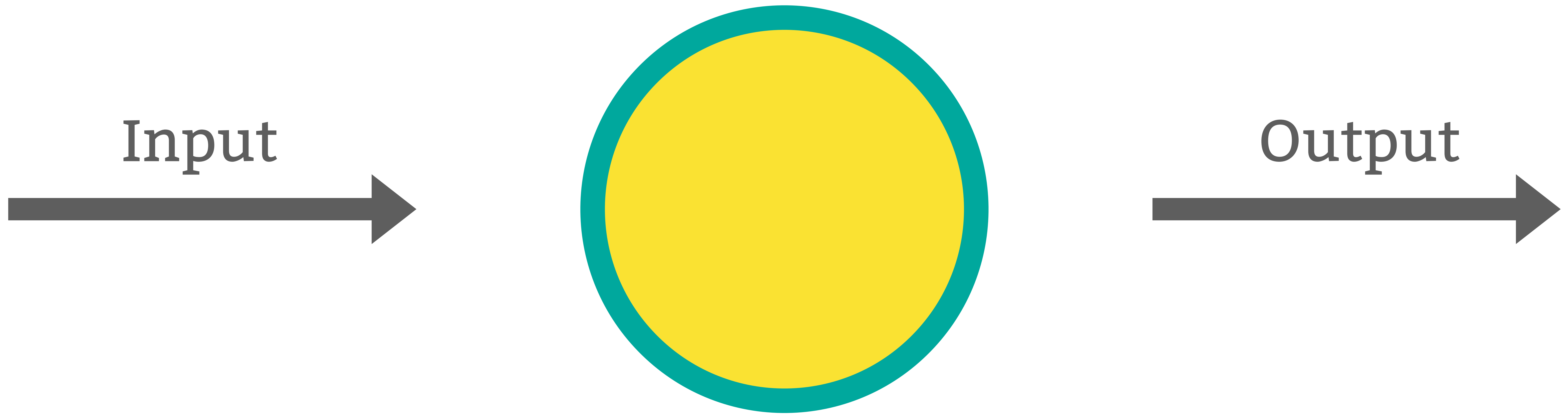
Functional Core, Imperative Shell



🟩 Imperative / Stateful

🟨 Functional / Stateless

Functional Core, Imperative Shell



🟩 Imperative / Stateful

🟨 Functional / Stateless

Takeaways

- Make illegal states unrepresentable

Takeaways

- Make illegal states unrepresentable
- Make side-effects visible in the type system

Takeaways

- Make illegal states unrepresentable
- Make side-effects visible in the type system
- Work with values/expressions, not statements

Takeaways

- Make illegal states unrepresentable
- Make side-effects visible in the type system
- Work with values/expressions, not statements
- Delay evaluation as much as possible

Takeaways

- Make illegal states unrepresentable
- Make side-effects visible in the type system
- Work with values/expressions, not statements
- Delay evaluation as much as possible
- Use finite state machines