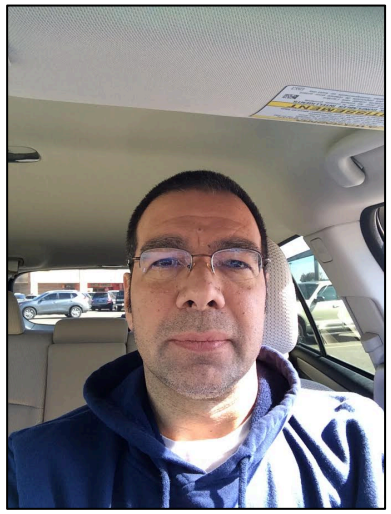
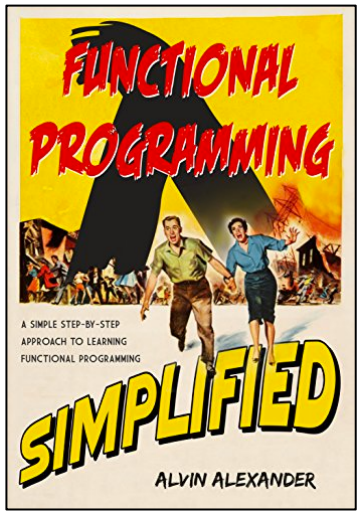
State Monad

learn how it works

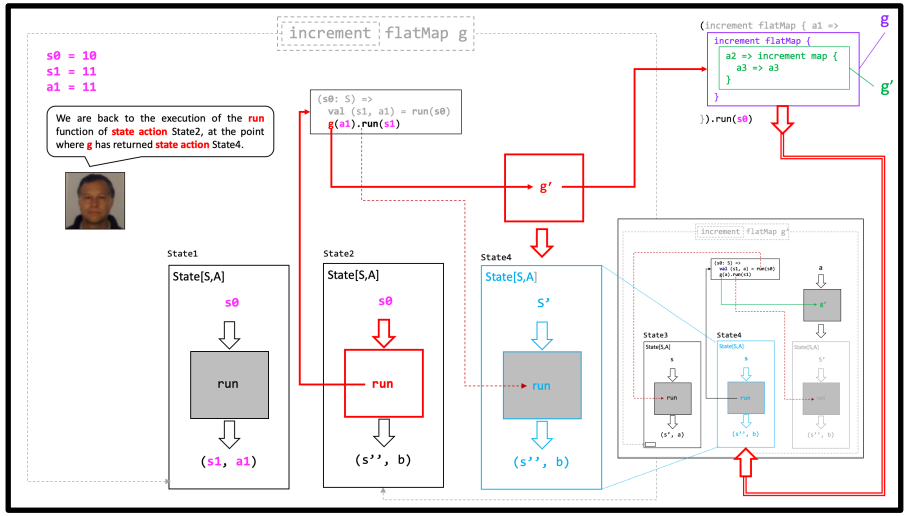
follow Alvin Alexander's example-driven build up to the State Monad

and then branch off into a detailed look at its inner workings



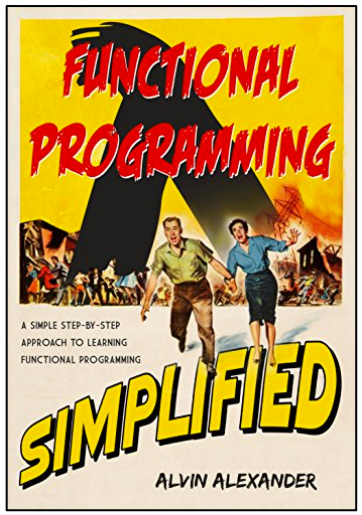


Alvin Alexander @alvinalexander



Philip Schwarz @philip_schwarz

Alvin Alexander @alvinalexander



82

An Introduction to Handling State

the **State monad** is a wrapper that makes the concept of "**state**" easier to work with in **for expressions**. The next few lessons will first demonstrate the **problems** of trying to work with **state** without a **State monad**, and then I'll show how the **State monad** helps to alleviate those **problems**.

If you don't happen to have a **State monad** laying around, you can still handle state in **Scala**/**FP**. The basic ideas are:

- First, create some sort of **construct to model the state of your application** at any moment in time. Typically this is a **case class**, but it doesn't have to be.
- Next, create a **"helper" function** that **takes** a) the **previous state** and b) some sort of **increment** or **"delta"** to that **state**. The function should **return a new state based on those two values**.

83

Handling State Manually

Imagine that you're on the first hole of a golf course, and you swing at a ball three times, with these results:

- The first ball goes **20** yards
- The second ball goes **15** yards
- The third swing is a "swing and a miss," so technically the ball goes **0** yards

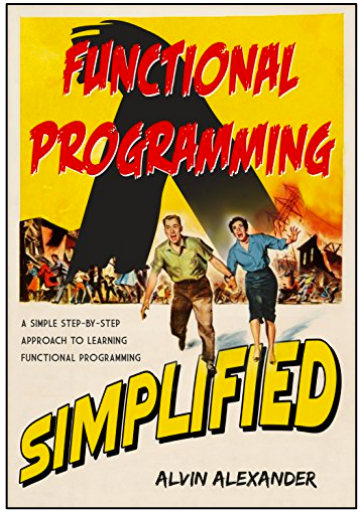
One way to model the **state** after each stroke is with a simple **case class** that **stores the cumulative distance of all my swings**:

```
case class GolfState(distance: Int)
```

Given that model, I can create a **"helper" function** named **nextStroke**. It takes the **previous** `GolfState` and the **distance of the next stroke** to return a **new** `GolfState`:

```
def nextStroke(previousState: GolfState, distanceOfNextHit: Int) =
  GolfState(previousState.distance + distanceOfNextHit)
```

Alvin Alexander @alvinalexander



83

Handling State Manually

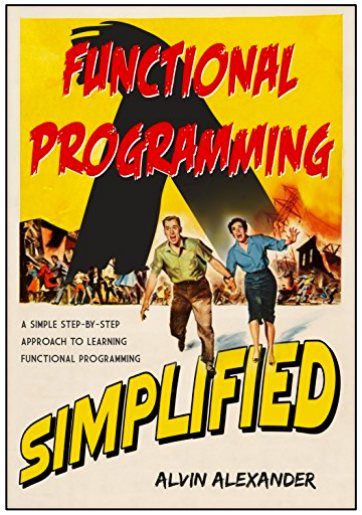
Now I can use those two pieces of code to create an application that models my three swings:

```
val state1 = GolfState(20)
val state2 = nextStroke(state1, 15)
val state3 = nextStroke(state2, 0)
println(state3) //prints "GolfState(35)"
```

The first three lines simulate my three swings at the ball. The last line of code prints the final golf **state** as `GolfState(35)`, meaning that the total distance of my swings is 35 yards.

This code won't win any awards — it's **repetitive and error-prone** — but it does show how **you have to model changing state in a Scala/FP application with immutable state variables**. (In the following lessons I show **how to improve on this situation by handling state in a for expression**.)

Alvin Alexander @alvinalexander



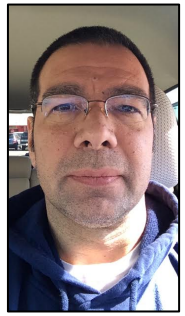
84

Getting State Working in a for Expression

Knowing that I wanted to get my code working in a **for expression**, **I attempted to create my own `State` monad**. I modeled my efforts after the `Wrapper` and `Debuggable` classes I shared about 10-20 lessons earlier in this book. Starting with that code, and dealing with only `Int` values, I created the following **first attempt at a `State` monad**:

```
/**
  * this is a simple (naive) attempt at creating a State monad
  */
case class State(value: Int) {

  def flatMap(f: Int => State): State = {
    val newState = f(value)
    State(newState.value)
  }

  def map(f: Int => Int): State = State(f(value))
}
```

This code is very similar to the `Wrapper` and `Debuggable` classes I created earlier in this book, so please see those lessons if you need a review of how this code works.



With this **`State` monad** in hand, I can write the following **for expression** to model my golf game:

```
val result = for {
  a <- State(20)
  b <- State(a + 15) // manually carry over 'a'
  c <- State(b + 0)  // manually carry over 'b'
} yield c

println(s"result: $result") // prints "State(35)"
```

The **good news** about this code is:

- It shows another example of a wrapper class that implements **flatMap** and **map**
- As long as I just need `Int` values, **this code lets me use a concept of state in a for expression**

Unfortunately, **the bad news is that I have to manually carry over values like a and b** (as shown in the comments), and **this approach is still cumbersome and error-prone**. Frankly, the only improvement I've made over the previous lesson is that **I now have "something" working in a for expression**.

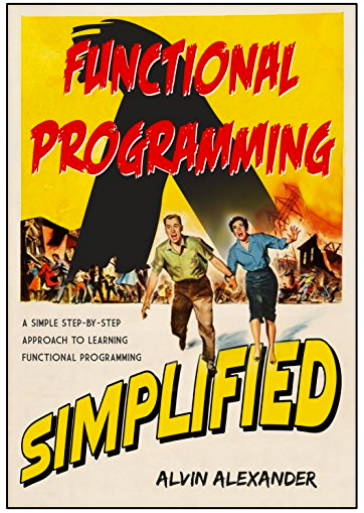
What I need is a better State monad.

While it is not necessary for the purposes of this slide deck, if you want to know more about the `Debuggable` class mentioned in the previous slide then see the following

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/writer-monad-for-logging-execution-of-functions

Alvin Alexander @alvinalexander



85

Handling My Golfing State with a State Monad

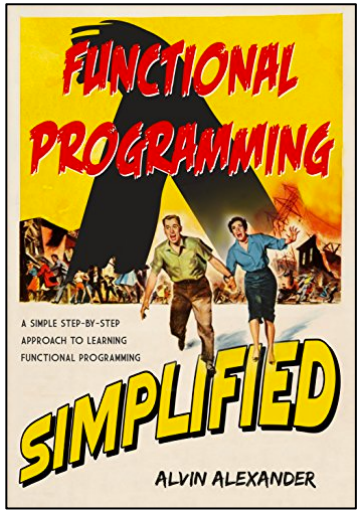
Fortunately some other people worked on this problem long before me, and they created a better `State` **monad** that lets me handle the problem of my three golf strokes like this:

```
val stateWithNewDistance: State[GolfState, Int] = for {
  _             <- swing(20)
  _             <- swing(15)
  totalDistance <- swing(0)
} yield totalDistance
```

Unlike the code in the previous lesson, notice that **there's no need to manually carry values over from one line in the for expression to the next line**. **A good `State` monad handles that bookkeeping for you**. In this lesson I'll show what you need to do to make this **for expression** work.

With a properly-written **State monad** I can write an application to simulate my golf game like this

Alvin Alexander @alvinalexander



85

Handling My Golfing State with a State Monad

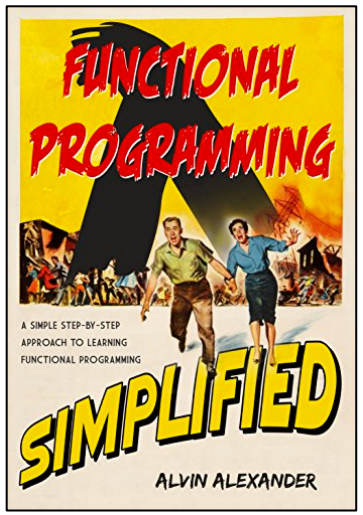
I'll explain this code in the remainder of this lesson.

```
object Golfing3 extends App {

  case class GolfState(distance: Int)

  def swing(distance: Int): State[GolfState, Int] =
    State { (s: GolfState) =>
      val newAmount = s.distance + distance
      (GolfState(newAmount), newAmount)
    }

  val stateWithNewDistance: State[GolfState, Int] =
    for {
                    _ <- swing(20)
                    _ <- swing(15)
      totalDistance <- swing(0)
    } yield totalDistance

  // initialize a `GolfState`
  val beginningState = GolfState(0)

  // run/execute the effect. …
  val result: (GolfState, Int) =
    stateWithNewDistance.run(beginningState)

  println(s"GolfState: ${result._1}") //GolfState(35)
  println(s"Total Distance: ${result._2}") //35
}
```

@philip_schwarz

Alvin spends the rest of chapter 85 explaining that code and then in chapter 86 he shows us the code for the **`State` monad**.

Alvin Alexander @alvinalexander



86

The State Monad Source Code

Until this point I treated the **State monad** code as a black box: I asked you to use it as though it already existed in the **Scala** libraries, just like you use `String`, `List`, and hundreds of other classes without thinking about how they're implemented.

My reason for doing this is that **the State code is a little complicated**. You have to be a real master of **for expressions** to be able to write a **State monad** that works like this.

Note: A "master of **for expressions**" is a goal to shoot for!

In a way, the **State monad** just implements **map** and **flatMap** methods, so it's similar to the `Wrapper` and `Debuggable` classes I created previously. But it also takes those techniques to another level by using generic types, by-name parameters, and anonymous functions in several places.

Here is the source code for the **State monad** I used in the previous lesson.

In this code the generic type `S` stands for "state," and then `A` and `B` are generic type labels, as usual.

In this first version of this book I'm not going to attempt to fully explain that code, but **I encourage you to work with it and modify it until you understand how it works**.

I decided not to show you the code for the **State monad** yet (see the next slide for why).

As we just saw, **Alvin** said that

- the **`State` monad**'s code is a little complicated
- He is not fully explaining the code in the first edition of his book
- The best way to understand the code is to work with it and modify it until we understand how it works

So instead of explaining **Alvin**'s code that uses the **`State` monad**, and instead of showing you straight away the **`State` monad** code he presented in the book, I am going to have a go at deriving the code for the **`State` monad** based on (a) how **Alvin**'s code uses the **monad** and (b) the known properties of a **monad**.

@philip_schwarz

As we can see below in **FPiS**, there are three minimal sets of primitive combinators that can be used to implement a **monad**:

- **unit**, **flatMap**
- unit, compose
- unit, map, join

I am going to pick the first set.

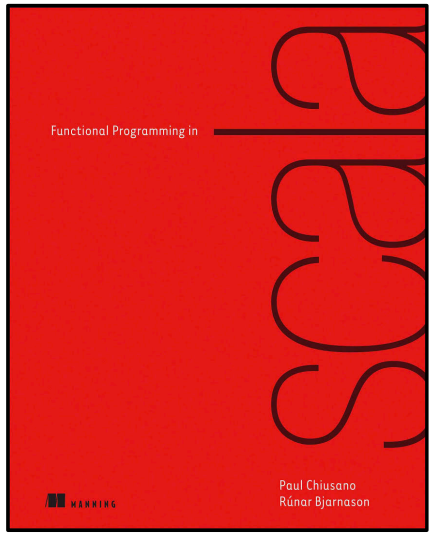
We've seen three minimal sets of primitive `Monad` combinators, and instances of `Monad` will have to provide implementations of one of these sets:

- `unit` and `flatMap`
- `unit` and `compose`
- `unit`, `map`, and `join`

And we know that there are two **monad** laws to be satisfied, associativity and identity, that can be formulated in various ways. So we can state plainly what a **monad** is :

> *A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.*

That's a perfectly respectable, precise, and terse definition. And if we're being precise, this is the only correct definition.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

So here is the trait a **monad** has to implement

```
trait Monad[F[_]] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
}
```

e.g. here is a **monad** instance for **Option**

```
val optionMonad: Monad[Option] = new Monad[Option] {
  override def unit[A](a: => A): Option[A] = Some(a)
  override def flatMap[A, B](ma: Option[A])(f: A => Option[B]): Option[B] =
    ma match {
      case None => None
      case Some(a) => f(a)
    }
}
```

In **Scala**, the **unit** function of each **monad** has a different name: the **Option monad** calls it **Some**, the **Either monad** calls it **Right**, the **List monad** calls it **List**, etc.

and here is an example of using the **Option monad**

```
val maybeName = optionMonad.unit("Fred")
val maybeSurname = optionMonad.unit("Smith")

val maybeFullName =
  optionMonad.flatMap(maybeName){ name =>
    optionMonad.flatMap(maybeSurname){ surname =>
      optionMonad.unit(s"$name $surname")
    }
  }

assert(maybeSomeFullName == optionMonad.unit("Fred Smith"))
```

flatMap is used to bind a variable to a **pure** value in an **effectful monadic** box/context and **unit** is used to **lift** a **pure** value into an **effectful monadic** box/context.

Every **monad** is also a **Functor** because the **map** function of **Functor** can be defined in terms of the **unit** and **flatMap** functions of **Monad**

```
trait Functor[F[_]] {
  def map[A, B](ma: F[A])(f: A => B): F[B]
}
```

```
trait Monad[F[_]] extends Functor[F] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  override def map[A, B](ma: F[A])(f: A => B): F[B] =
    flatMap(ma)(a => unit(f(a)))
}
```

To **map** a function **f** over a **monadic context** we use **flatMap** to first apply **f** to the **pure** value found in the **monadic context** and then use **unit** to **lift** the result into an unwanted **monadic context** that **flatMap** strips out.

Now that our **Monad** definition includes a **map** function we can simplify our sample usage of the **Option monad**

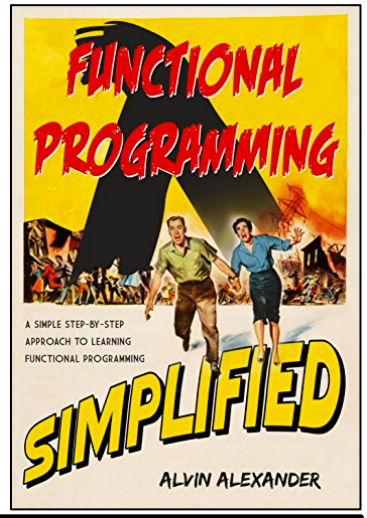
```
val maybeFullName =
  optionMonad.flatMap(maybeName){ name =>
    optionMonad.flatMap(maybeSurname){ surname =>
      optionMonad.unit(s"$name $surname")
    }
  }
```

```
val maybeFullName =
  optionMonad.flatMap(maybeName){ name =>
    optionMonad.map(maybeSurname){ surname =>
      s"$name $surname"
    }
  }
```

In the above, we defined the **Functor** and **Monad** traits ourselves because there are no such traits in **Scala**.

What about in plain **Scala**, i.e. without rolling our own **Functor** and **Monad** traits and without using the **Functor** and **Monad** traits provided by an FP library like **Scalaz** or **Cats**, how is a **Monad** implemented?

Alvin Alexander @alvinalexander



60

Recap: Option -> flatMap -> for

When speaking casually, some people like to say that any **Scala** class that implements **map** and **flatMap** is a **monad**. While that isn't 100% true, it's in the ballpark of truth.

As **Gabriele Petronella** wrote in a Stack Overflow post:

> "The only thing **monads** are relevant for, from a **Scala** language perspective, is the ability of being used in a **for-comprehension**."

By this he means that while **monads** are defined more formally in a language like **Haskell**, **in Scala there is no base Monad trait to extend; all you have to do is implement map and flatMap so your class can be used as a generator in for expressions**.

```
case class Foo[A](n:A)

val fooTwo = Foo(2)
val fooThree = Foo(3)
val fooFive = Foo(5)
```

Alvin is referring to the fact that in **Scala** I can take any class, e.g. **Foo** here on the left hand side, and turn it into a **monad** by giving it a **map** function and a **flatMap** function, as shown here on the right hand side

```
case class Foo[A](n:A) {
  def map[B](f: A => B): Foo[B] = Foo(f(n))
  def flatMap[B](f: A => Foo[B]): Foo[B] = f(n)
}
```

```
val result =
  fooTwo flatMap { x =>
    fooThree map { y =>
      x + y
    }
  }

assert(result == fooFive)
```

And as a result, instead of having to write code that looks like the example on the left hand side, I can write code using the **syntactic sugar** of the **for comprehension** on the right hand side, i.e. the compiler translates (**desugars**) the code on the right hand side to the code on the left hand side.

```
val result =
  for {
    x <- fooTwo
    y <- fooThree
  } yield x + y


assert(result == fooFive)
```

```scala
assert(
  (for {
        name <- Some("Fred")
     surname <- Some("Smith")
  } yield s"$name $surname")
  == Some("Fred Smith")
)
```

@philip_schwarz

```scala
assert(
  Some("Fred").flatMap{ name =>
    Some("Smith").flatMap{ surname =>
      Some(s"$name $surname")
    }
  }
  == Some("Fred Smith")
)
```

I imagine that because in Scala each monad calls the unit function differently, the compiler is not able to desugar the above code to the code on the left hand side, which would only require the monad to have a flatMap function and a unit function (called Some in this case), so instead monads in Scala *have to have* a map function because the compiler can then translate the above code to the code on the right hand side.

```scala
assert(
  Some("Fred").flatMap{ name =>
    Some("Smith").map{ surname =>
      s"$name $surname"
    }
  }
  == Some("Fred Smith")
)
```

Alvin's first attempt at a **State monad** was a **case class** and we'll use the same approach.

We can see from **Alvin**'s code here on the right that `State[GolfState, Int]` has a parameter that is a function from `GolfState` to `(GolfState, Int)` and that the parameter is called **run**.

`GolfState` is the type of the **state** whose current value is consumed by the **monad**'s **run** function and whose next (successor) value is computed by the **run** function. `Int` is the type of the result (the distance) that is computed by the **run** function.

So let's start coding the **State monad**.

We'll use `S` for the type of the **state** that is both consumed and produced by the **monad**'s **run** function. We'll use `A` for the type of the result that is computed by the **monad**'s **run** function using the **state**.

```
case class State[S, A](run: S => (S, A))
```

```
object Golfing3 extends App {

  case class GolfState(distance: Int)

  def swing(distance: Int): State[GolfState, Int] =
    State { (s: GolfState) =>
      val newAmount = s.distance + distance
      (GolfState(newAmount), newAmount)
    }

  val stateWithNewDistance: State[GolfState, Int] =
    for {
                  _ <- swing(20)
                  _ <- swing(15)
      totalDistance <- swing(0)
    } yield totalDistance

  // initialize a `GolfState`
  val beginningState = GolfState(0)

  // run/execute the effect. …
  val result: (GolfState, Int) =
    stateWithNewDistance.run(beginningState)

  println(s"GolfState: ${result._1}") //GolfState(35)
  println(s"Total Distance: ${result._2}") //35
}
```

In **Alvin**'s program we can see four **State monad** instances being created.

Three instances are created by the three calls to the **swing** function in the **for comprehension**.

The fourth instance is the result of the whole **for comprehension**.

We can see where the **run** function of the fourth instance is being invoked.

What about the **run** functions of the other three instances? When are they invoked? There are no references to them anywhere!!!

To answer that we look at the desugared version of the **for comprehension**:

```
swing(20) flatMap { _ =>
  swing(15) flatMap { _ =>
    swing(0) map { totalDistance =>
      totalDistance
    }
  }
}
```

The only usage of the **State monad** instances created by the **swing** function consists of calls to the **map** and **flatMap** functions of the instances. Their **run** functions are not invoked. So it must be their **map** and **flatMap** functions that invoke their **run** functions.

```
object Golfing3 extends App {

  case class GolfState(distance: Int)

  def swing(distance: Int): State[GolfState, Int] =
    State { (s: GolfState) =>
      val newAmount = s.distance + distance
      (GolfState(newAmount), newAmount)
    }

  val stateWithNewDistance: State[GolfState, Int] =
    for {
                    _ <- swing(20)
                    _ <- swing(15)
      totalDistance <- swing(0)
    } yield totalDistance

  // initialize a `GolfState`
  val beginningState = GolfState(0)

  // run/execute the effect. …
  val result: (GolfState, Int) =
    stateWithNewDistance.run(beginningState)

  println(s"GolfState: ${result._1}") //GolfState(35)
  println(s"Total Distance: ${result._2}") //35
}
```

Let's turn to the task of implementing the **map** and **flatMap** functions of the **State monad**.

Let's start with **map**, the easiest of the two.

```
case class State[S, A](run: S => (S, A)) {

  def map[B](f: A => B): State[S, B] = ???

}
```

map returns a new **State monad** instance

```
def map[B](f: A => B): State[S,B] =
  State { s =>
    ???
  }
```

①

the instance's **run** function returns an S and a B

```
def map[B](f: A => B): State[S,B] =
 State { s =>
    val s1 = ???
    val b = ???
    (s1,b)
  }
```

②

If we have an A then we can get a B by calling **f**

```
def map[B](f: A => B): State[S,B] =
 State { s =>
    val a = ???
    val s1 = ???
    val b = f(a)
    (s1,b)
  }
```

③

we can get S and A by calling our **run** function

```
def map[B](f: A => B): State[S,B] =
  State { s =>
    val result = run(s)
    val a = result._2
    val s1 = result._1
    val b = f(a)
    (s1,b)
  }
```

④

simplify using pattern matching

```
def map[B](f: A => B): State[S,B] =
  State { s =>
    val (s1,a) = run(s)
    val b = f(a)
    (s1,b)
  }
```

⑤

inline **b**

```
def map[B](f: A => B): State[S,B] =
  State { s =>
    val (s1, a) = run(s)
    (s1, f(a))
  }
```

⑥

Now let's implement **flatMap**.

@philip_schwarz

```
case class State[S, A](run: S => (S, A)) {

  def map[B](f: A => B): State[S, B] =
    State { s =>
      val (s1, a) = run(s)
      (s1, f(a))
    }

  def flatMap[B](g: A => State[S, B]): State[S, B] = ???
}
```

flatMap returns a new **State monad** instance

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    ???
  }
```

①

the instance's run function returns an S and a B

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val s1 = ???
    val b = ???
    (s1,b)
  }
```

②

we can get an S and a B if we have a **State**[S,B]

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val state: State[S, B] = ???
    val result = state.run(s)
    val s1 = result._1
    val b = result._2
    (s1, b)
  }
```

③

simplify

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val state: State[S, B] = ???
    state.run(s)
  }
```

④

If we have an A then we can get a **State**[S,B] by calling **f**

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val a = ???
    val state: State[S, B] = f(a)
    state.run(s)
  }
```

⑤

We can get an A by calling our own **run** function

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val (s1,a) = run(s)
    val state: State[S, B] = f(a)
    state.run(s)
  }
```

⑥

inline **state** and use **s1** in second call to run

```
def flatMap[B](f:A=>State[S,B]):State[S,B] =
  State { s =>
    val (s1,a) = run(s)
    f(a).run(s1)
  }
```

⑦

Remember how earlier we saw that every **monad** is also a **Functor** because the **map** function of **Functor** can be defined in terms of the **unit** and **flatMap** functions of **Monad**?

Let's implement the **unit** function of the **State monad** so that we can simplify the implementation of the **map** function.

unit returns a new **State monad** instance

```
def unit[S,A](a: =>A): State[S,A] =
  State { s =>
    ???
  }
```

①

the instance's **run** function returns an S and a A

```
def unit[S,A](a: =>A): State[S,A] =
 State { s =>
    val s1 = ???
    val a = ???
    (s1,a)
  }
```

②

we can just use the A and S that we are given

```
def unit[S,A](a: =>A): State[S,A] =
  State { s =>
    (s, a)
  }
```

③

Now we can simplify the **map** function as follows

```
def map[B](f: A => B): State[S,B] =
  State { s =>
    val (s1,a) = run(s)
    f(a).run(s1)
  }
```

```
def map[B](f: A => B): State[S,B] =
  flatMap( a => unit(f(a)) )
```

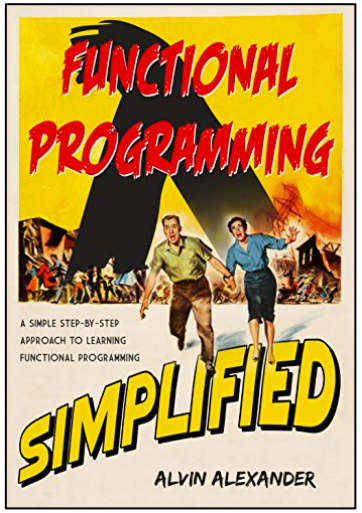
Below is the **State monad** implementation we ended up with and on the right hand side you can see the implementation presented by **Alvin** in his book.

```
object State {

  def unit[S, A](a: => A): State[S, A] =
    State { s =>
      (s, a)
    }
}

import State._
case class State[S, A](run: S => (S, A)) {

  def map[B](f: A => B): State[S, B]
    flatMap(a => unit(f(a)))

  def flatMap[B](f: A => State[S, B]): State[S, B] =
    State { s =>
      val (s1, a) = run(s)
      f(a).run(s1)
    }
}
```

The only real difference is that the **unit** function is called **point** and its parameter is passed by name rather than by value.

A minor difference is that the function parameter of **flatMap** is called **g** whereas that of **map** is called **f**.

Alvin Alexander @alvinalexander

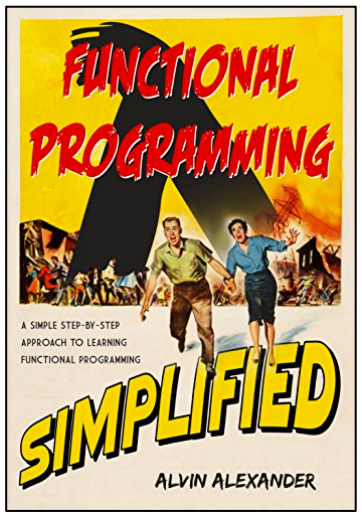


86

The State Monad Source Code

```
case class State[S, A](run: S => (S, A)) {

  def flatMap[B](g: A => State[S, B]): State[S, B] = State { (s0: S) =>
      val (s1, a) = run(s0)
      g(a).run(s1)
    }

  def map[B](f: A => B): State[S, B] = flatMap( a => State.point(f(a)) )
}

object State {
  def point[S, A](v: A): State[S, A] = State(run = s => (s, v))
}
```

Alvin Alexander @alvinalexander



86

The State Monad Source Code

John A De Goes
@jdegoes

Don't be intimidated!

I'll also make two other points at this time. First, **I doubt that anyone wrote a State monad like this on their first try**. **I'm sure it took several efforts before someone figured out how to get what they wanted in** a **for expression**.

Second, while **this code can be hard to understand in one sitting**, I've looked at some of the source code inside the **Scala** collections classes, and there's code in there that's also hard to grok. (Take a look at the sorting algorithms and you'll see what I mean.)

Personally, **the only way I can understand complex code like this is to put it in a Scala IDE and then modify it until I make it my own**.

Where State comes from

I believe the original version of this **State** code came from this Github URL:

- github.com/jdegoes/lambdaconf-2014-introgame

As the text at that link states, "This repository contains the material for **Introduction to Functional Game Programming with Scala**, held at **LambdaConf 2014** in Boulder, Colorado."

While I find a lot of that material to be hard to understand without someone to explain it (such as at a conference session), **Mr. De Goes created his own `State` monad for that training session, and I believe that was the original source for the `State` monad I just showed**.

Much of the inspiration for this book comes from attending that conference and thinking, "I have no idea what these people are talking about."

@philip_schwarz

Deriving the implementation of **map** and **flatMap** starting from their signatures is doable in that the types in play pretty much dictate the implementation.

But to truly understand how the **State monad** works I had to really study those methods and visualise what they do.

So in the rest of this slide deck I will have a go at visualising, in painstaking detail, how the **map** and **flatMap** functions operate.

In order to eliminate any unnecessary source of distraction I will do this in the context of an example that is even simpler than **Alvin**'s golfing example.

I'll use the **State monad** to implement a trivial counter that simply gets incremented on each state transition. That way it will be even easier to concentrate purely on the essentials of how the **State monad** works.

```
def increment: State[Int, Int] =
  State { (count: Int) =>
    val nextCount = count + 1
    (nextCount, nextCount)
  }
```

In **Alvin**'s golfing example the **state** consisted of a `GolfState case class` that wrapped a **distance** of type `Int`. In our minimal **counter** example the **state** is simply going to be an `Int`: we are not going to bother wrapping it in a `case class`.

So instead of our `State[S, A]` instances being of type `State[GolfState, Int]`, they are going to be of type `State[Int, Int]`.

And instead of the **run** functions `S => (S, A)` of our `State[S, A]` instances being of type `GolfState => (GolfState, Int)`, they are going to be of type `Int =>(Int,Int)`

```
val tripleIncrement: State[Int, Int] =
  for {
    _      <- increment
    _      <- increment
    result <- increment
  } yield result
```

Each `<-` binds to a variable the `A` result of invoking the `S=>(S,A)` **run** function of a `State[S,A]` instance created by an invocation of the `increment` function. Since our `State[S,A]` instances have type `State[Int,Int]`, each `<-` binds an `Int` count result to a variable.

In the first two cases the variable is `_` because we don't care about intermediate counter values. In the third case the variable is called **result** because the final value of the counter is what we care about.

What the **for comprehension** does is compose three `State[Int,Int]` instances whose **run** functions each perform a single increment, into a single `State[Int,Int]` instance whose **run** function performs three increments.

```
val initialState = 10

val (finalState, result) =
  tripleIncrement.run(initialState)

assert( finalState == 13 )
assert( result == 13 )
```

We invoke the **run** function of the composite `State[Int,Int]` instance with an initial count **state** of 10.

The function increments the count three times and returns both the final **state** at the end of the increments and the value of the counter, which is the same as the **state**.

One more slide to further explain why the values that the **for comprehension** binds to variables _, _, and result below are indeed the A values obtained by invoking the S=>(S,A) **run** functions of the **State**[S, A] instances created by invoking the **increment** function.

If we look at the signature of the **map** and **flatMap** functions again, we are reminded that they both take a function whose domain is type A.

```
def map[B](f: A => B): State[S, B]
def flatMap[B](f: A => State[S, B]): State[S, B]
```

So the anonymous lambda functions you see passed to **map** and **flatMap** below, on the right hand side, have domain A, which we indicated by naming their variables **a1**, **a2** and **a3**.

In the case of this counter example, the intermediate A results, i.e. the first two, are of no interest and so we have greyed out the first two variables, which are unused.

original

```
val tripleIncrement: State[Int, Int] =
  for {
    _      <- increment
    _      <- increment
    result <- increment
  } yield result
```

desugared

```
val tripleIncrement: State[Int, Int] =
  increment flatMap { _ =>
    increment flatMap { _ =>
      increment map { result =>
        result
      }
    }
  }
```

variables renamed

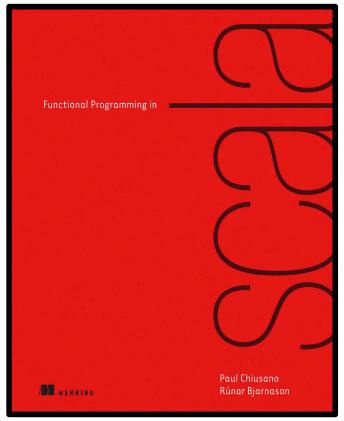
```
val tripleIncrement: State[Int, Int] =
  increment flatMap { a1 =>
    increment flatMap { a2 =>
      increment map { a3 =>
        a3
      }
    }
  }
```

I got a bit tired of repeating the words '**State monad** instance', so in what follows I will instead take inspiration from **FPiS** and say '**state action**'.

```
type State[S,+A] = S => (A,S)
```

Here **State is short for computation that carries some state along, or state action, state transition, or even statement** (see the next section). We might want to write it as its own class, wrapping the underlying function like this:

```
case class State[S,+A](run: S => (A,S))
```



Functional Programming in Scala

In order to fully understand how the **State monad** works, in the rest of this slide deck we are going to examine and visualise, in painstaking detail, how the following desugared **for comprehension** is executed:

```
increment flatMap { a1 =>
    increment flatMap { a2 =>
      increment map { a3 =>
        a3
      }
    }
  }
```

We are going to be using the **State monad** code in **Alvin**'s book (see below) because this part of the slide deck came before the part in which I had a go at deriving the **State monad** code.

```
case class State[S, A](run: S => (S, A)) {

  def flatMap[B](g: A => State[S, B]): State[S, B] =
    State { (s0: S) =>
      val (s1, a) = run(s0)
      g(a).run(s1)
    }

  def map[B](f: A => B): State[S, B] =
    flatMap( a => State.point(f(a)) )
}

object State {
  def point[S, A](v: A): State[S, A] = State(run = s => (s, v))
}
```

Actually that is not the full story.

What the desugared **for comprehension** does is compose the **state actions** returned by the invocations of the `increment` function into a new **composite state action**.

Once the desugared **for comprehension** has produced this **composite state action**, we'll want to execute its **run** function, passing in an integer that represents the initial state of a counter.

```
increment flatMap { a1 =>
    increment flatMap { a2 =>
      increment map { a3 =>
        a3
      }
    }
  }.run(10)
```

The **run** function should return (13, 13) i.e. a **final state** of 13 and a **result value** of 13.

@philip_schwarz

We want to evaluate this expression:

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

The function passed to **flatMap** is called **g**:

```
def flatMap[B](g: A => State[S, B]): State[S, B]
```

So to simplify the expression to be evaluated, let's take the function passed to the first **flatMap** invocation

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

g

and let's replace it with the name **g**

```
increment flatMap g
```

In the next slide we are going to get started first evaluating `increment flatMap g` and then invoking the **run** function of the resulting **state action**.

We are going to go very slow to make sure everything is easy to understand.

It is going to take almost 60 slides.

If you get tired, just grasp the main ideas and jump to the last six slides for some final remarks/observations.

when invoked, both **increment** and **flatMap** return a **state action**.

```
increment flatMap g
```

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

function **g** is passed as a parameter to **flatMap**.

a

function **g** takes parameter **a** of type **A**.

increment

flatMap

g

when invoked, function **g** returns a **state action**.

State[S,A]

s

run

(s', a)

State[S,A]

s

run

(s'', b)

State[S,A]

S'

run

(s'', b)

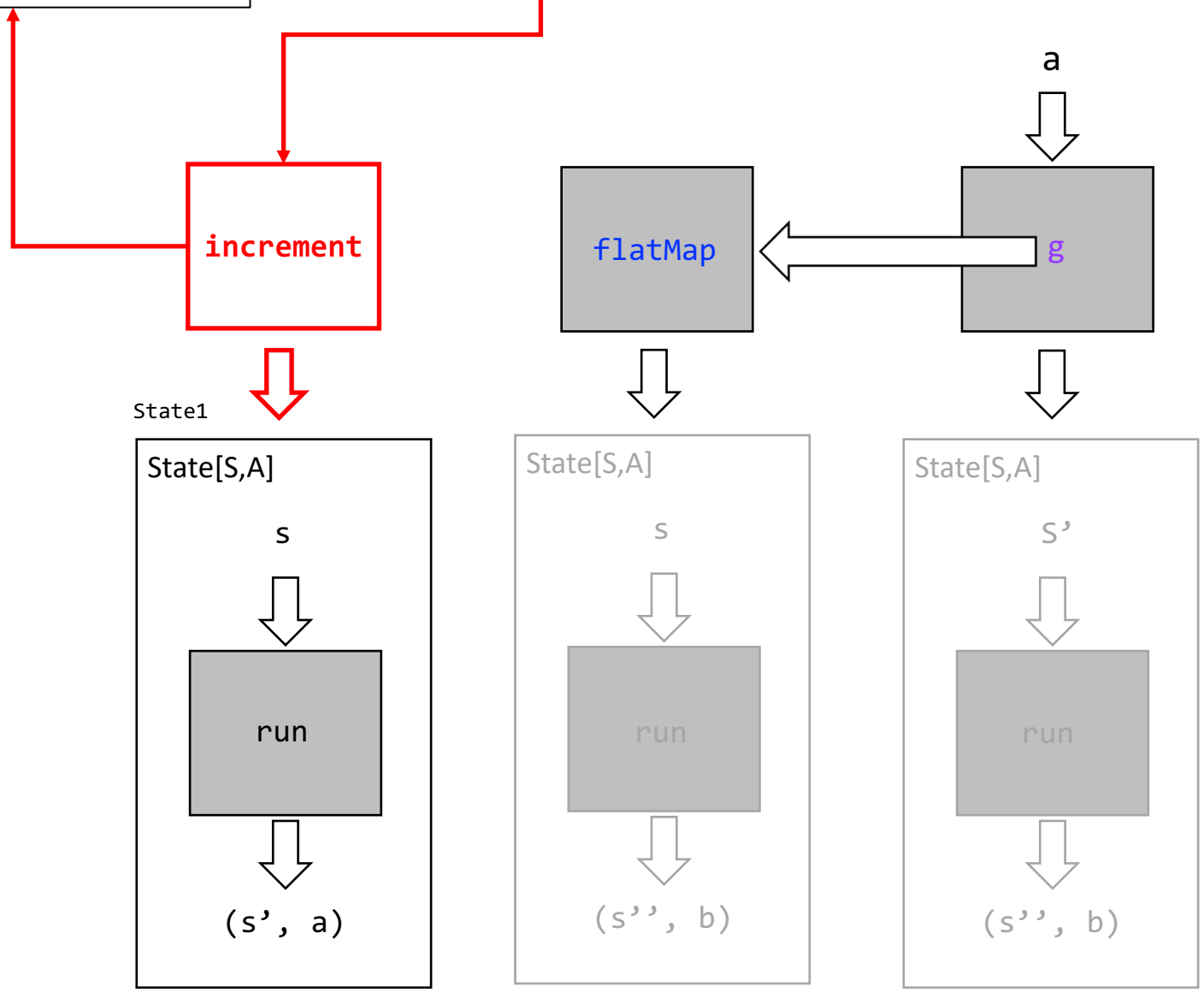
The first step in evaluating our expression is the invocation of the **increment** function. See next slide.

```
def increment: State[Int, Int] =
  State { (count: Int) =>
    val nextCount = count + 1
    (nextCount, nextCount)
  }
```

increment flatMap g

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

g

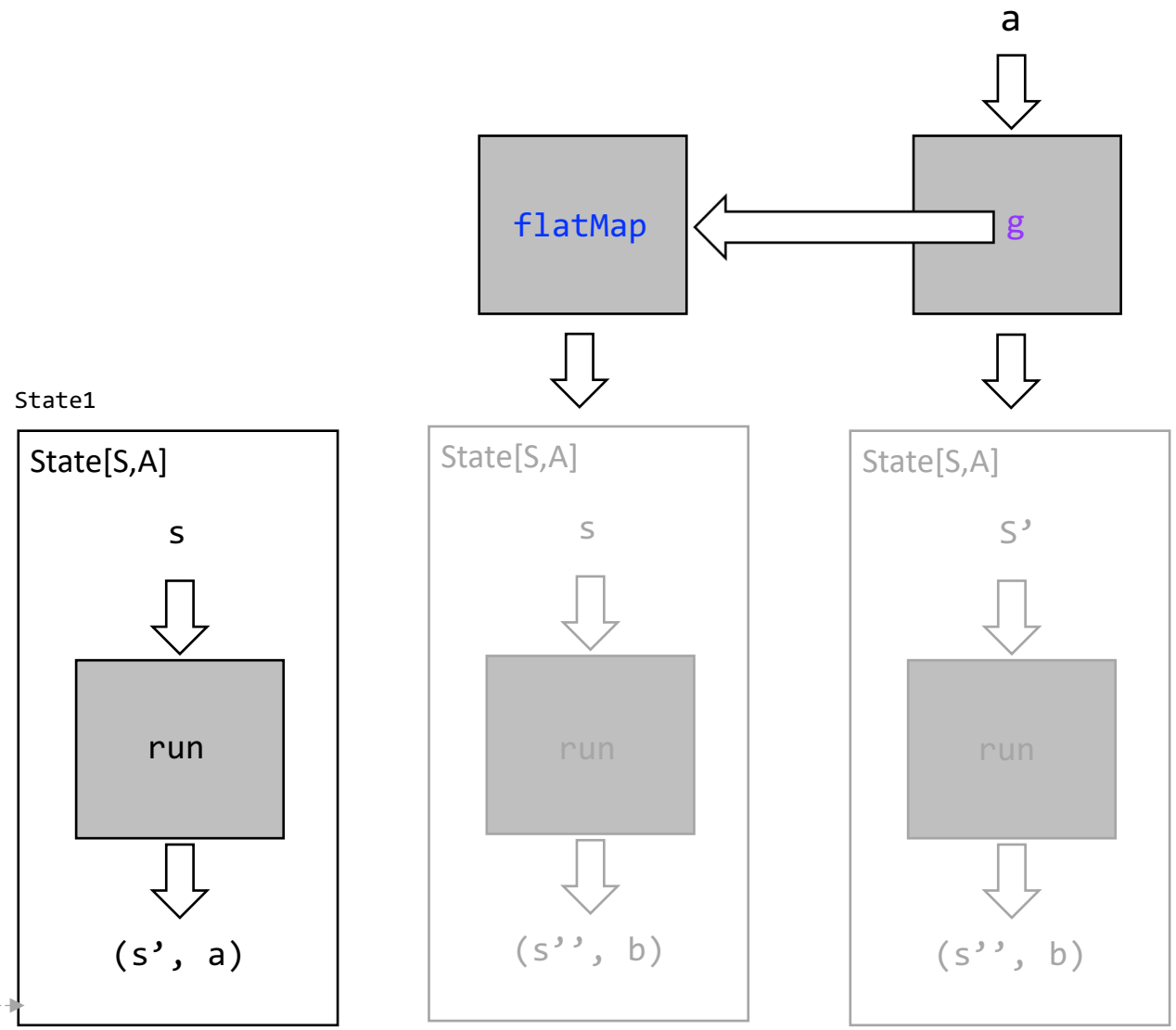


The **increment** function returns **state action** State1.

increment flatMap g

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

g

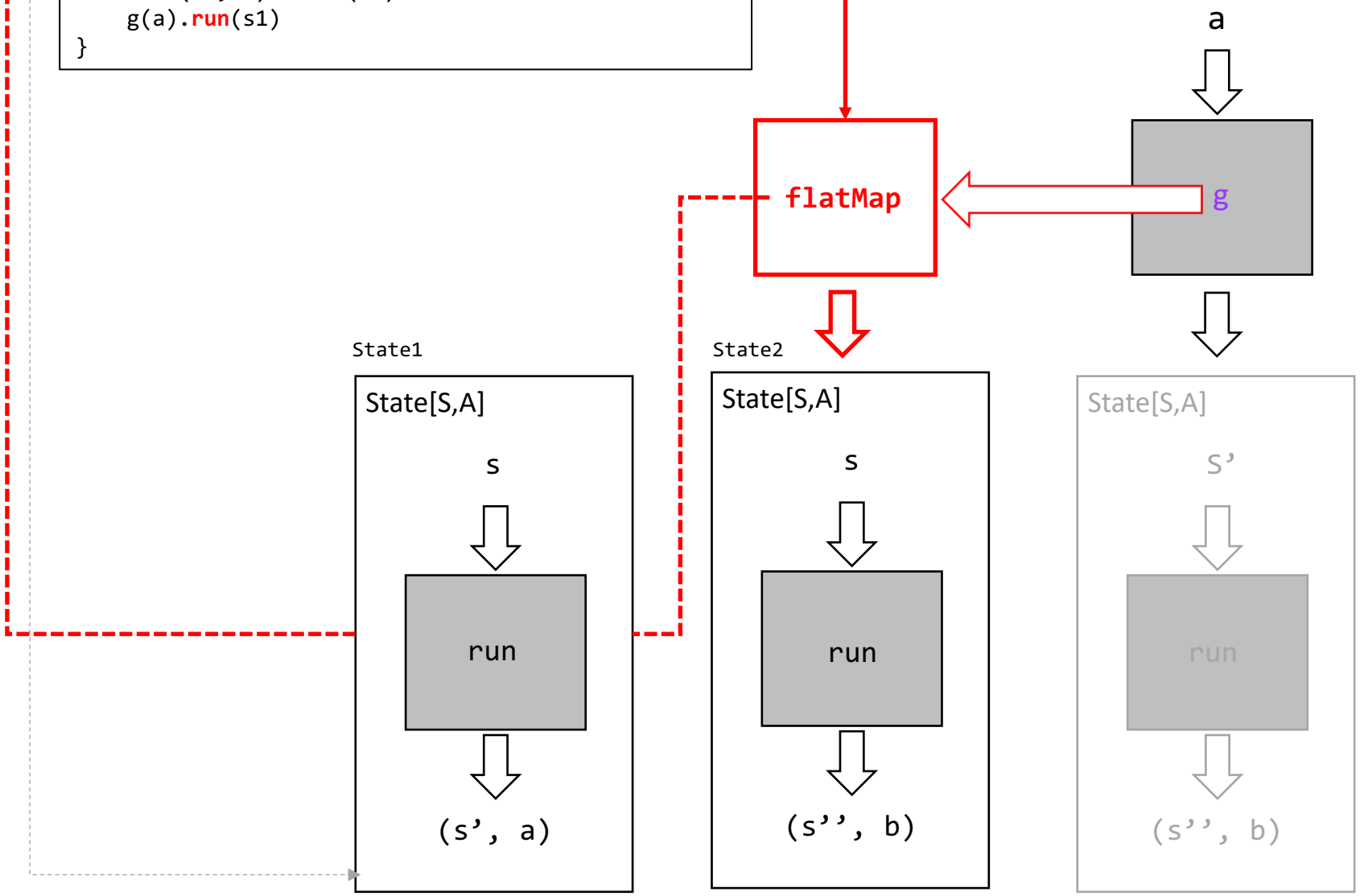


The increment subexpression evaluated to new **state action** State1. The next step is to call the **flatMap** function of State1, passing in **g** as a parameter.

@philip_schwarz

increment flatMap g

```
def flatMap[B](g: A => State[S, B]): State[S, B] =
  State { (s0: S) =>
    val (s1, a) = run(s0)
    g(a).run(s1)
}
```




```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

g

Invoking State1's **flatMap** function produces new **state action** State2.

increment flatMap g

```
increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}
```

g

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g

State1

State[S,A]

s

run

(s', a)

State2

State[S,A]

s

run

(s'', b)

State[S,A]

S'

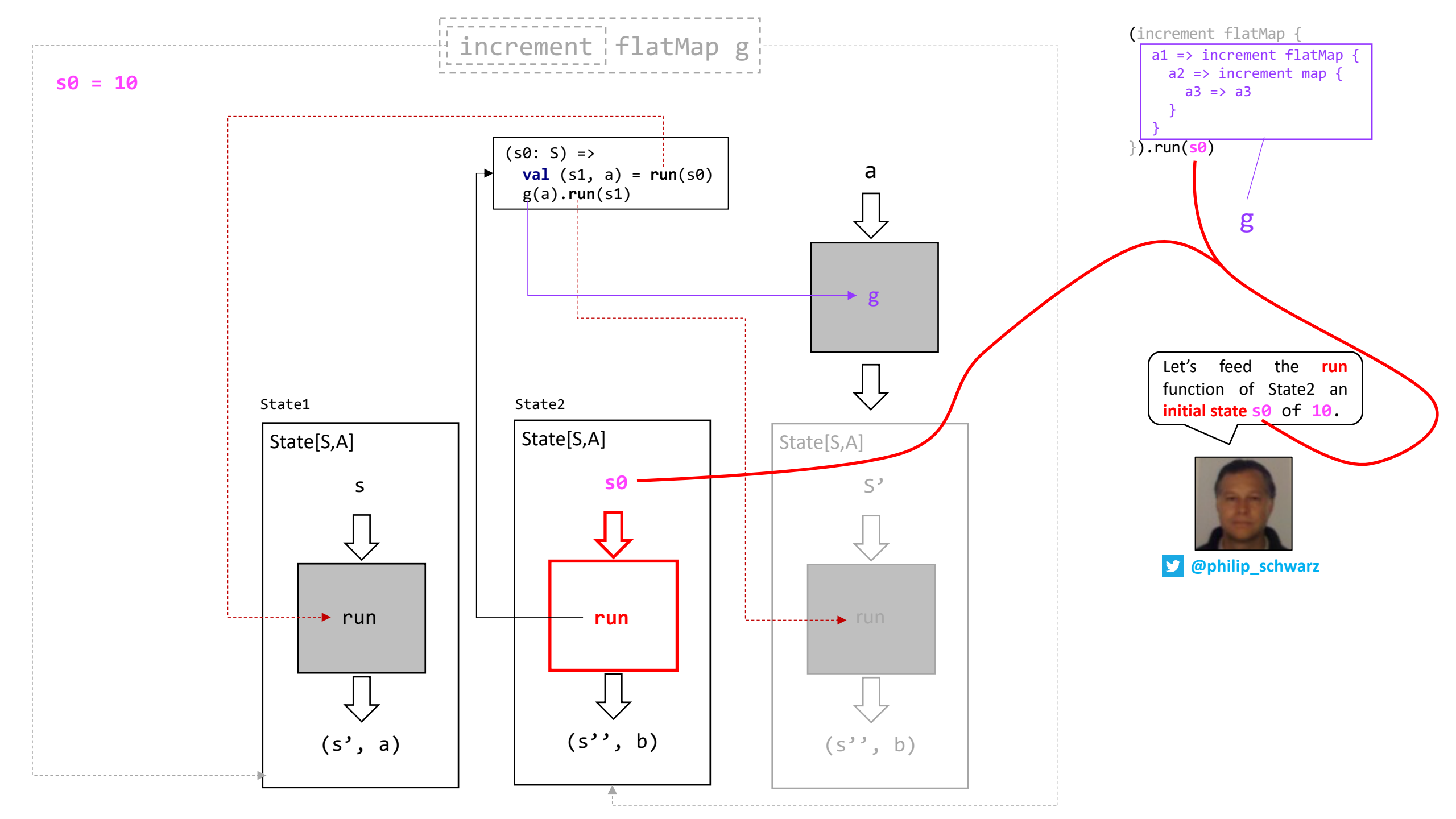
run

(s'', b)

So the value of the expression **increment flatMap g** is **state action** State2 and we are done!

Not quite. As we said before, this is not the whole story. Now that we have a composite **state action** we need to invoke its **run** function with an **Int** that represents the **initial state** of a counter.

We are going to start counting at **10** (see next slide).


increment flatMap g
s0 = 10
(s0: S) =>
val (s1, a) = run(s0)
g(a).run(s1)
a
g
State1
State[S,A]
s
run
(s', a)
State2
State[S,A]
s0
run
(s'', b)
State[S,A]
S'
run
(s'', b)
(increment flatMap {
a1 => increment flatMap {
a2 => increment map {
a3 => a3
}
}
}).run(s0)
g
Let's feed the run function of State2 an initial state s0 of 10.
@philip_schwarz

increment flatMap g

s0 = 10

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g

State1

State[S,A]

s

run

(s', a)

State2

State[S,A]

s0

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

Let's pass **state s0** into the body of State2's **run** function.

increment flatMap g

s0 = 10

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g

State1

State[S,A]

s

run

(s', a)

State2

State[S,A]

s0

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

Let's evaluate the body of State2's **run** function.

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

a

g

State1

State[S,A]

s0

run

(s1, a1)

```
(count: Int) =>
  val nextCount = count + 1
  (nextCount, nextCount)
```

State2

State[S,A]

s0

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

The first thing that the **run** function of State2 does is call the **run** function of State1, passing in state **s0**.

This returns (**s1**,**a1**) i.e. the **next state** and a **result** counter, both being **11**.

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

a

g

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s'', b)

State[S,A]

S'

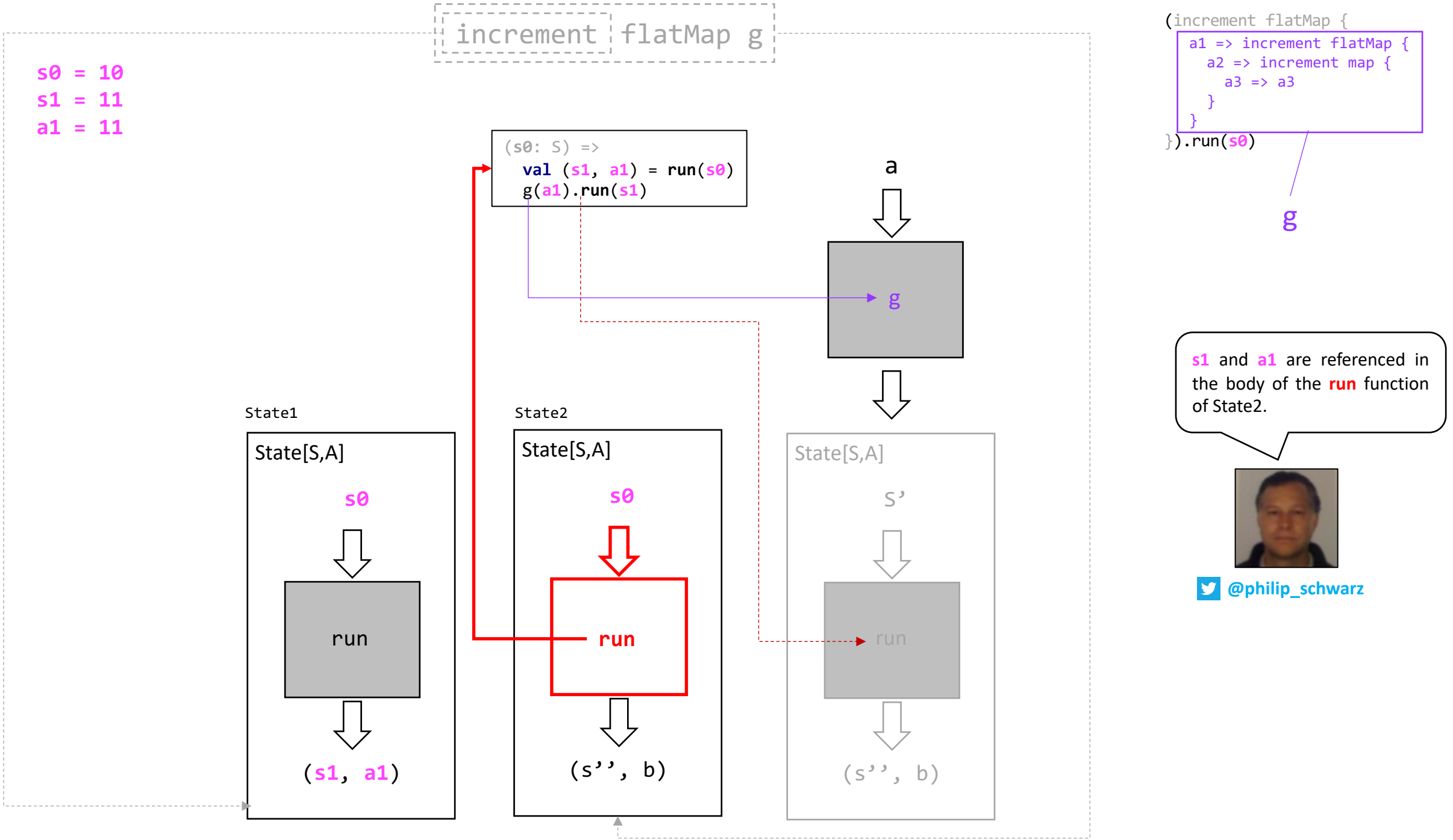
run

(s'', b)

```
(increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

The run function of State2 receives (s1, a1), the result of invoking the run function of State1.


increment flatMap g
s0 = 10
s1 = 11
a1 = 11
(s0: S) =>
val (s1, a1) = run(s0)
g(a1).run(s1)
a
g
State1
State[S,A]
s0
run
(s1, a1)
State2
State[S,A]
s0
run
(s'', b)
State[S,A]
S'
run
(s'', b)
(increment flatMap {
a1 => increment flatMap {
a2 => increment map {
a3 => a3
}
}
}).run(s0)
g
s1 and a1 are referenced in the body of the run function of State2.
@philip_schwarz

```
s0 = 10
s1 = 11
a1 = 11
```

increment flatMap g

```
(increment flatMap {
  a1 => increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

a

g

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

The next thing that the **run** function of State2 does is call function **g** with the **a1** value computed by State1’s **run** function.

In this simple example, in which the **State monad** is simply used to increment a counter a number of times, the **a1** and **a2** values in the **desugared for comprehension** are not actually used, and so we could rename them to _ if we so wished.

a3 on the other hand **_is_** used: it is returned as the result/value of the whole computation.

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

a1

g

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

g

The body of **g** consists of another case of **flatMapping** a function, **g’** say, over the result of invoking the **increment** function.

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

g

```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

g

The evaluation of **increment flatMap g'**, over the next 7 slides, will proceed in the same way as the evaluation of **increment flatMap g**, so feel free to fast-forward through it.

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s'', b)

State[S,A]

S'

run

(s'', b)

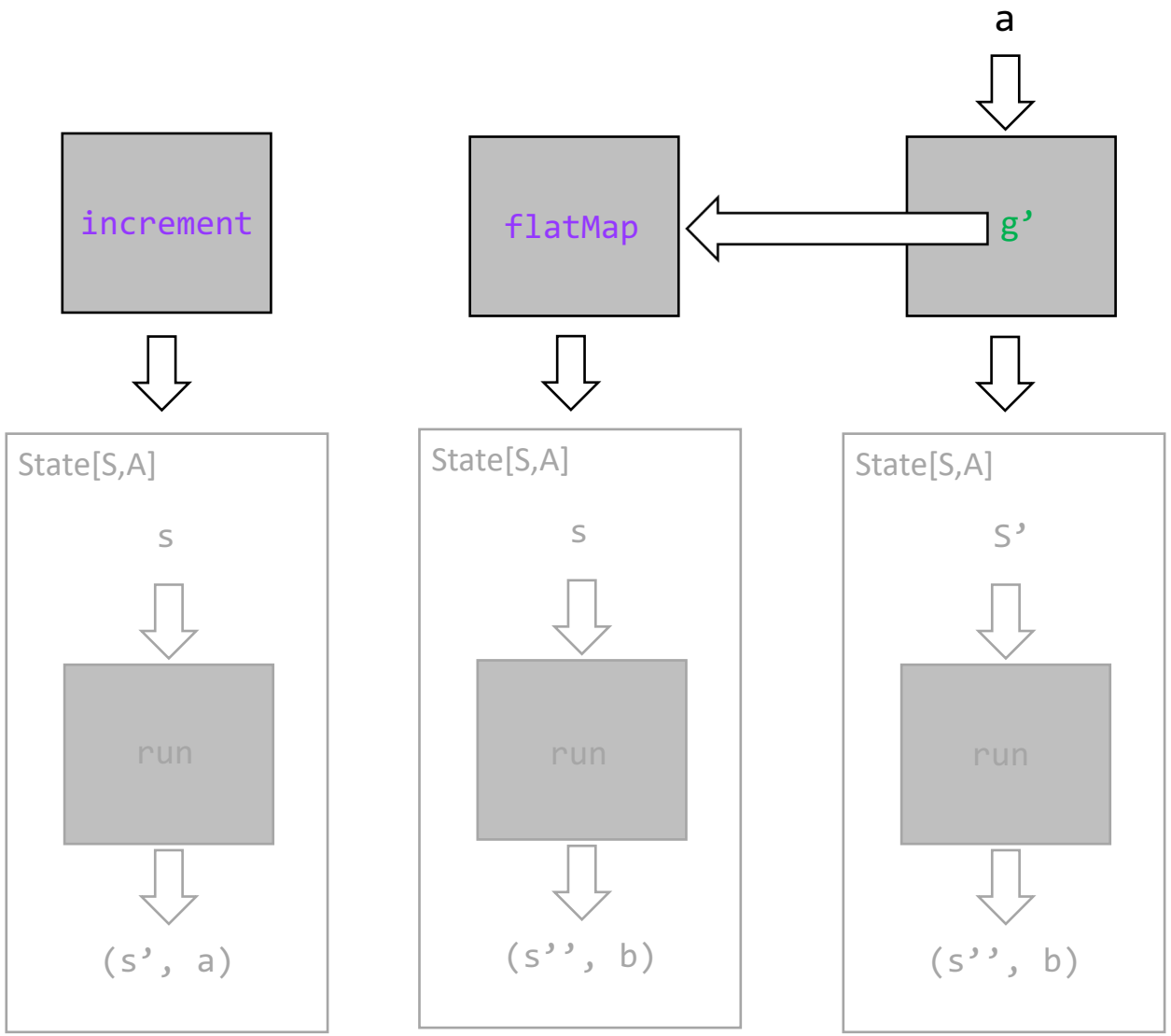
```
s0 = 10
s1 = 11
a1 = 11
```

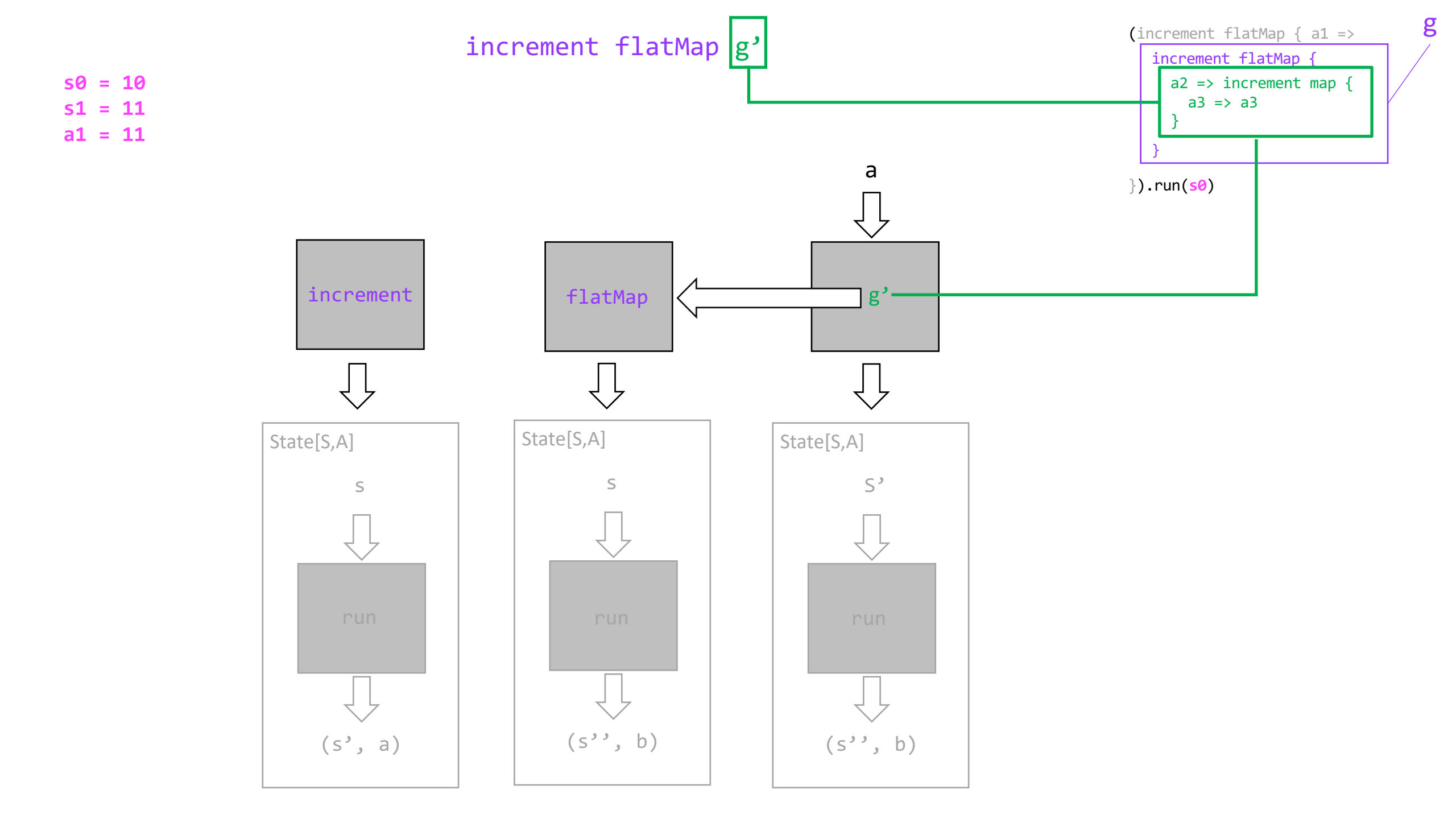
```
increment flatMap g'
```

```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'




s0 = 10
s1 = 11
a1 = 11
increment flatMap g’
(increment flatMap { a1 =>
increment flatMap {
a2 => increment map {
a3 => a3
}
}
}).run(s0)
g
a
increment
flatMap
g’
State[S,A]
s
run
(s’, a)
State[S,A]
s
run
(s’’, b)
State[S,A]
S’
run
(s’’, b)

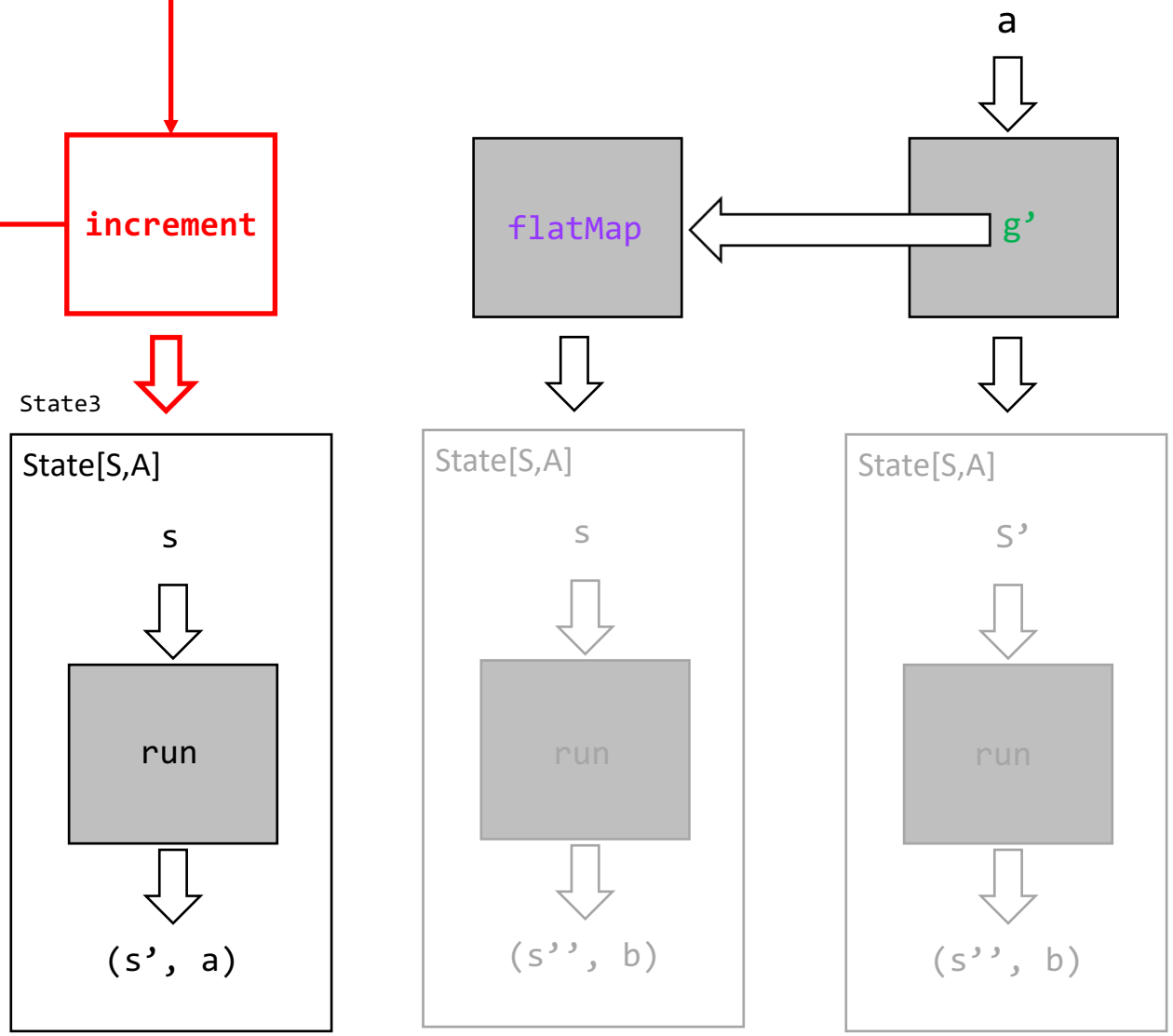
```
def increment: State[Int, Int] =
  State { (count: Int) =>
    val nextCount = count + 1
    (nextCount, nextCount)
  }
```

increment flatMap g’

```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’



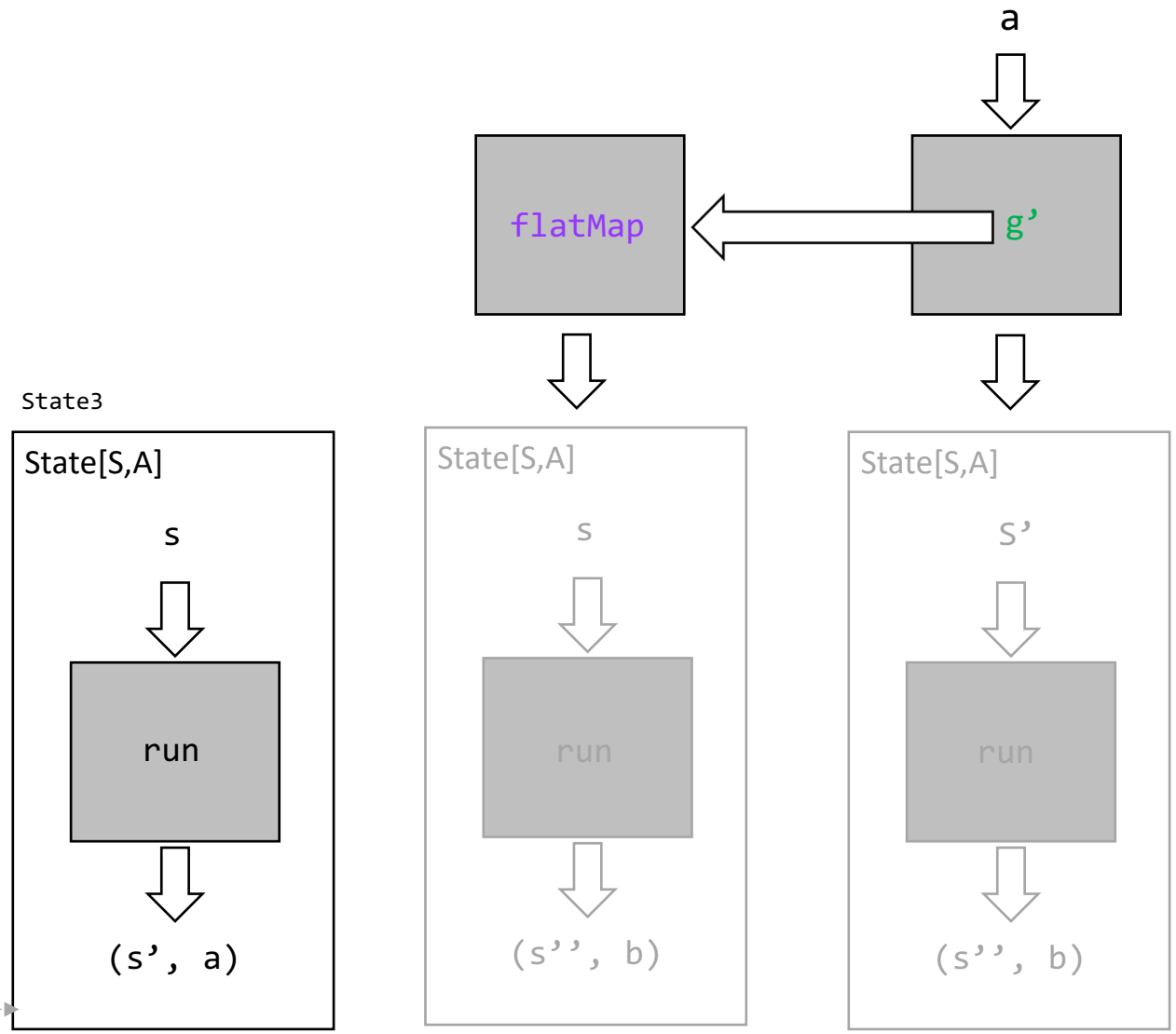
```
s0 = 10
s1 = 11
a1 = 11
```

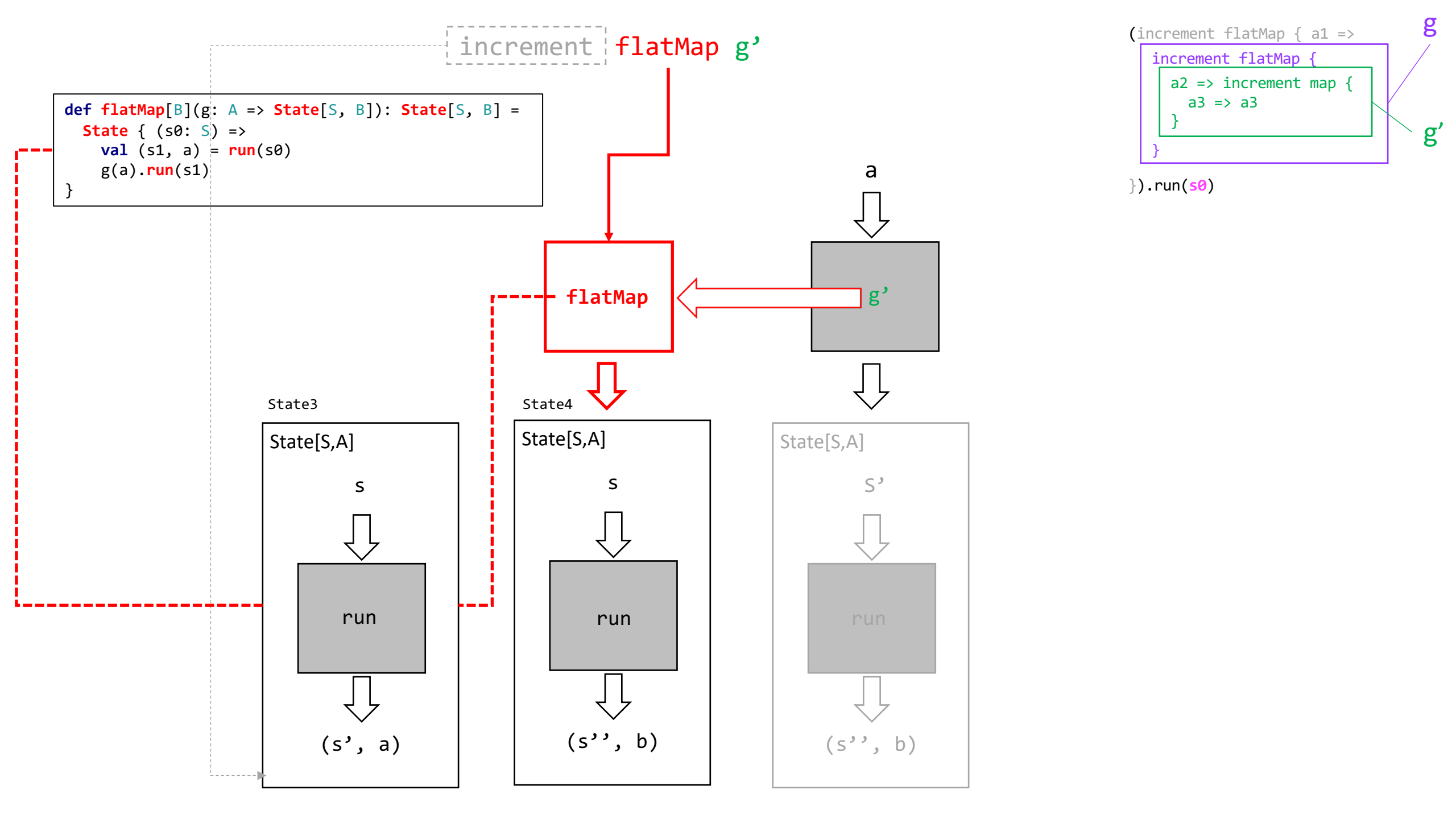
increment flatMap g’

```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’



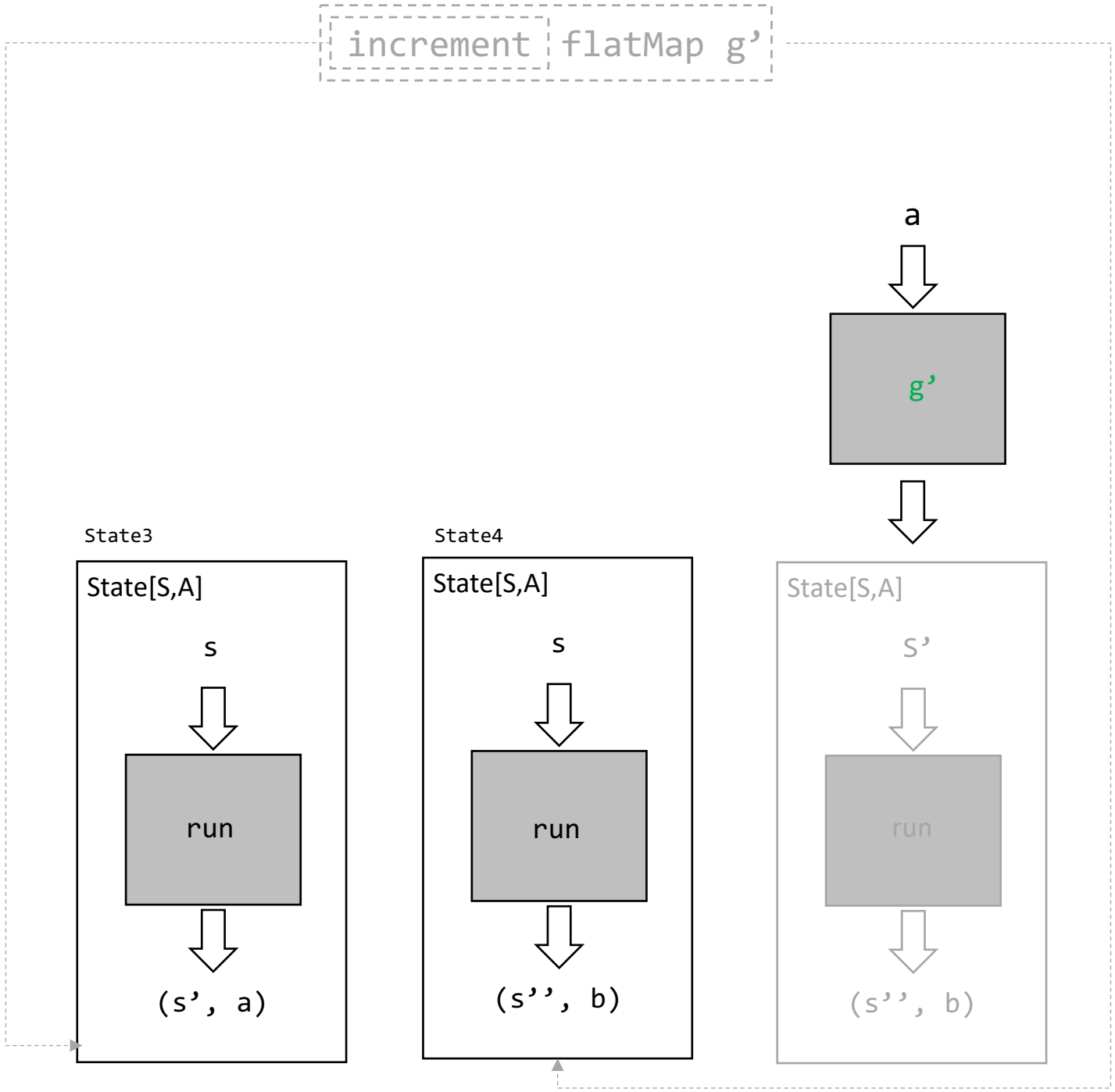

increment
flatMap g’
def flatMap[B](g: A => State[S, B]): State[S, B] =
State { (s0: S) =>
val (s1, a) = run(s0)
g(a).run(s1)
}
(increment flatMap { a1 =>
increment flatMap {
a2 => increment map {
a3 => a3
}
}
}).run(s0)
g
g’
a
flatMap
g’
State3
State[S,A]
s
run
(s’, a)
State4
State[S,A]
s
run
(s’’, b)
State[S,A]
S’
run
(s’’, b)

```
s0 = 10
s1 = 11
a1 = 11
```

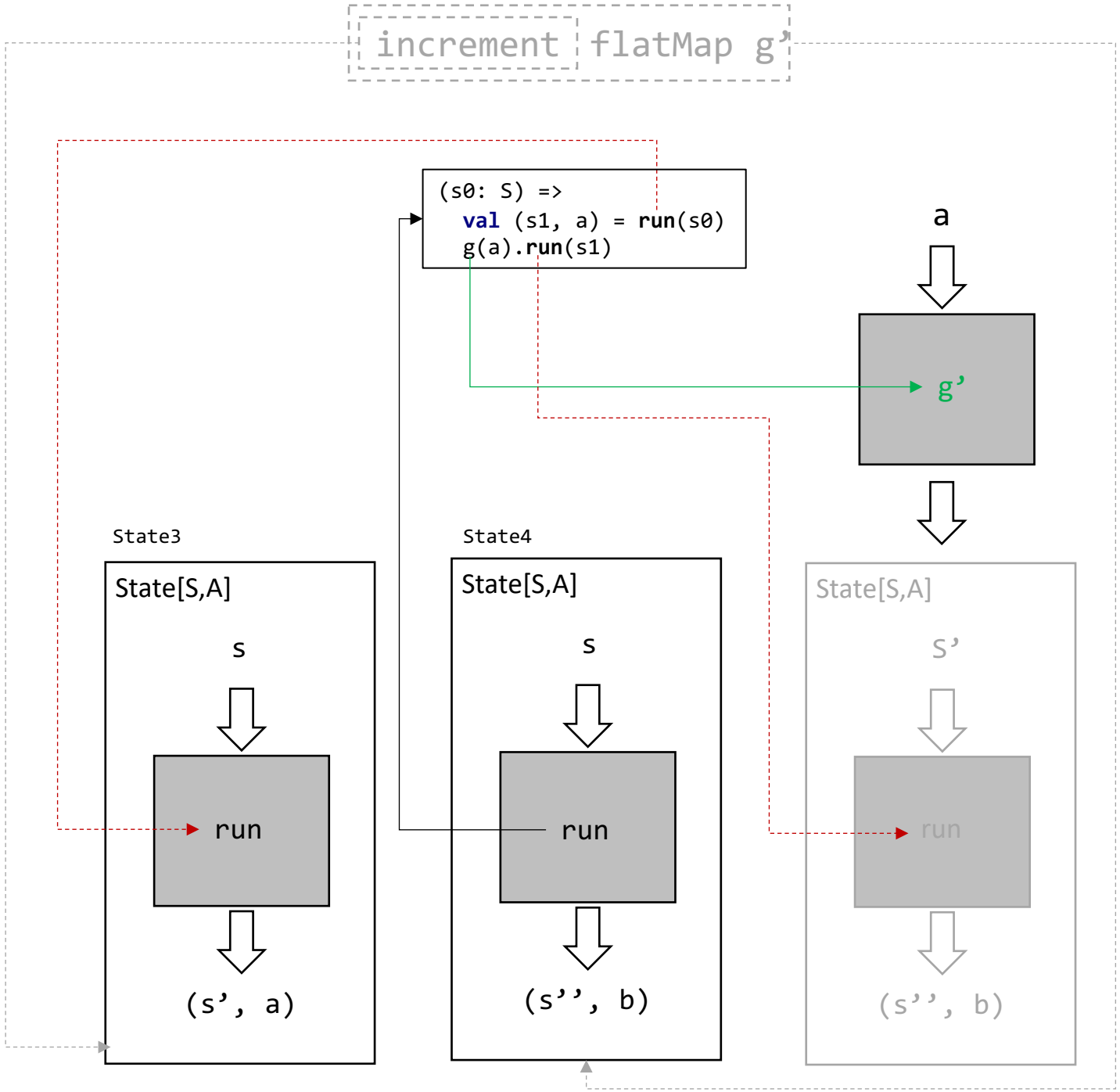
```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’



```
s0 = 10
s1 = 11
a1 = 11
```




```
(increment flatMap { a1 =>
  increment flatMap {
    a2 => increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

We have finished computing **increment flatMap g’** in the same way we had already computed **increment flatMap g**.

The result is **state action** State4. In the next slide we return to the execution of the **run** method of State2, which will now make use of State4.

@philip_schwarz

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

We are back to the execution of the **run** function of **state action** State2, at the point where **g** has returned **state action** State4.

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

g'

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s'', b)

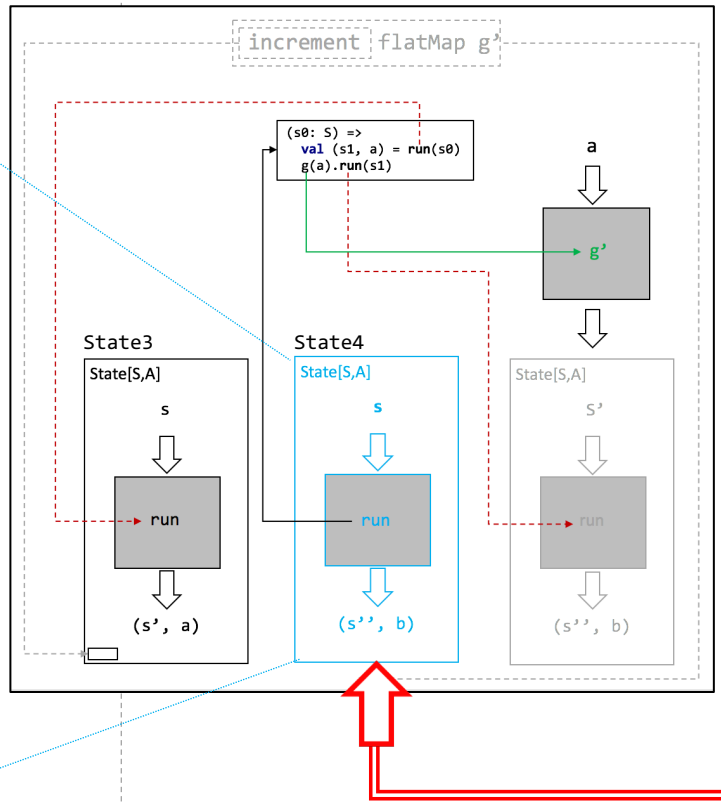
State4

State[S,A]

S'

run

(s'', b)



increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

In the next slide, the **run** function of **state action** State2 is going to invoke the **run** function of **state action** State4.

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

State1

State[S,A]

s0

run

(s1, a)

State2

State[S,A]

s0

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

increment flatMap g'

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g'

State3

State[S,A]

s

run

(s', a)

State4

State[S,A]

s

run

(s'', b)

State[S,A]

s'

run

(s'', b)

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
```

The **run** function of **state action** State4 is being invoked with the latest state **s1** as a parameter.

Turn to slide 83 if you want to skip to the point where the result of the invocation is returned.

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

State1

State[S,A]

s0

run

(s1, a)

State2

State[S,A]

s0

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

increment flatMap g'

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g'

State3

State[S,A]

s

run

(s', a)

State4

State[S,A]

s1

run

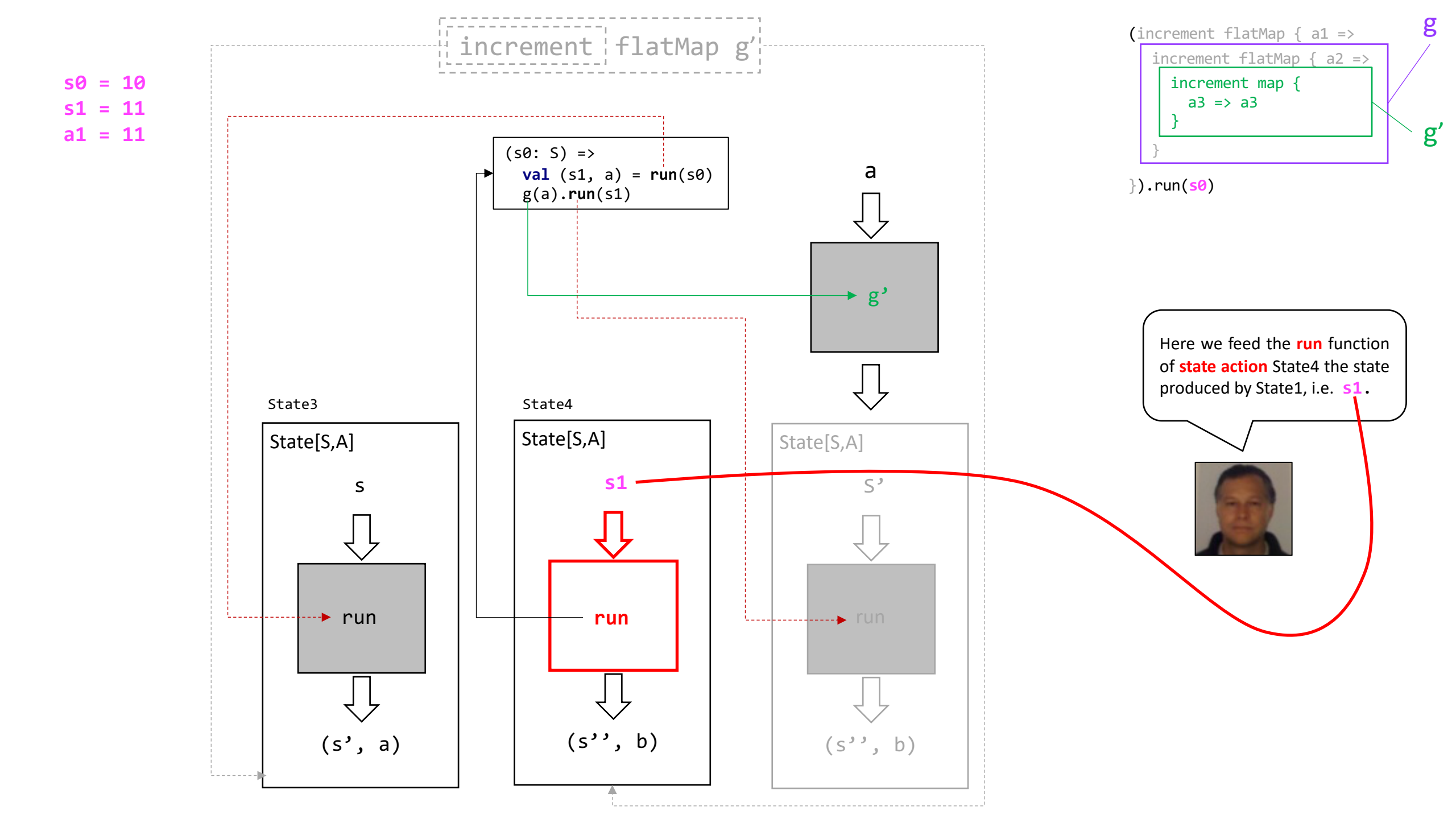
(s'', b)

State[S,A]

s'

run

(s'', b)


s0 = 10
s1 = 11
a1 = 11
increment flatMap g'
(s0: S) =>
val (s1, a) = run(s0)
g(a).run(s1)
a
g'
State3
State[S,A]
s
run
(s', a)
State4
State[S,A]
s1
run
(s'', b)
State[S,A]
S'
run
(s'', b)
(increment flatMap { a1 =>
increment flatMap { a2 =>
increment map {
a3 => a3
}
}
}).run(s0)
g
g'
Here we feed the run function of state action State4 the state produced by State1, i.e. s1.

increment flatMap g’

```
s0 = 10
s1 = 11
a1 = 11
```

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

a

g’

State3

State[S,A]

s

run

(s’, a)

State4

State[S,A]

s1

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

Let’s pass s1 into the body of State4’s run function.

@philip_schwarz

increment flatMap g’

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

a

g’

State3

State[S,A]

s1

run

(s2, a2)

```
(count: Int) =>
  val nextCount = count + 1
  (nextCount, nextCount)
```

State4

State[S,A]

s1

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

The first thing that the **run** function of State4 does is call the **run** function of State3, passing in state **s1**.

This returns (**s2**,**a2**) i.e. the **next state** and a **result** counter, both being **12**.

increment flatMap g’

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

a

g’

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

(s’’, b)

State[S,A]

S’

run

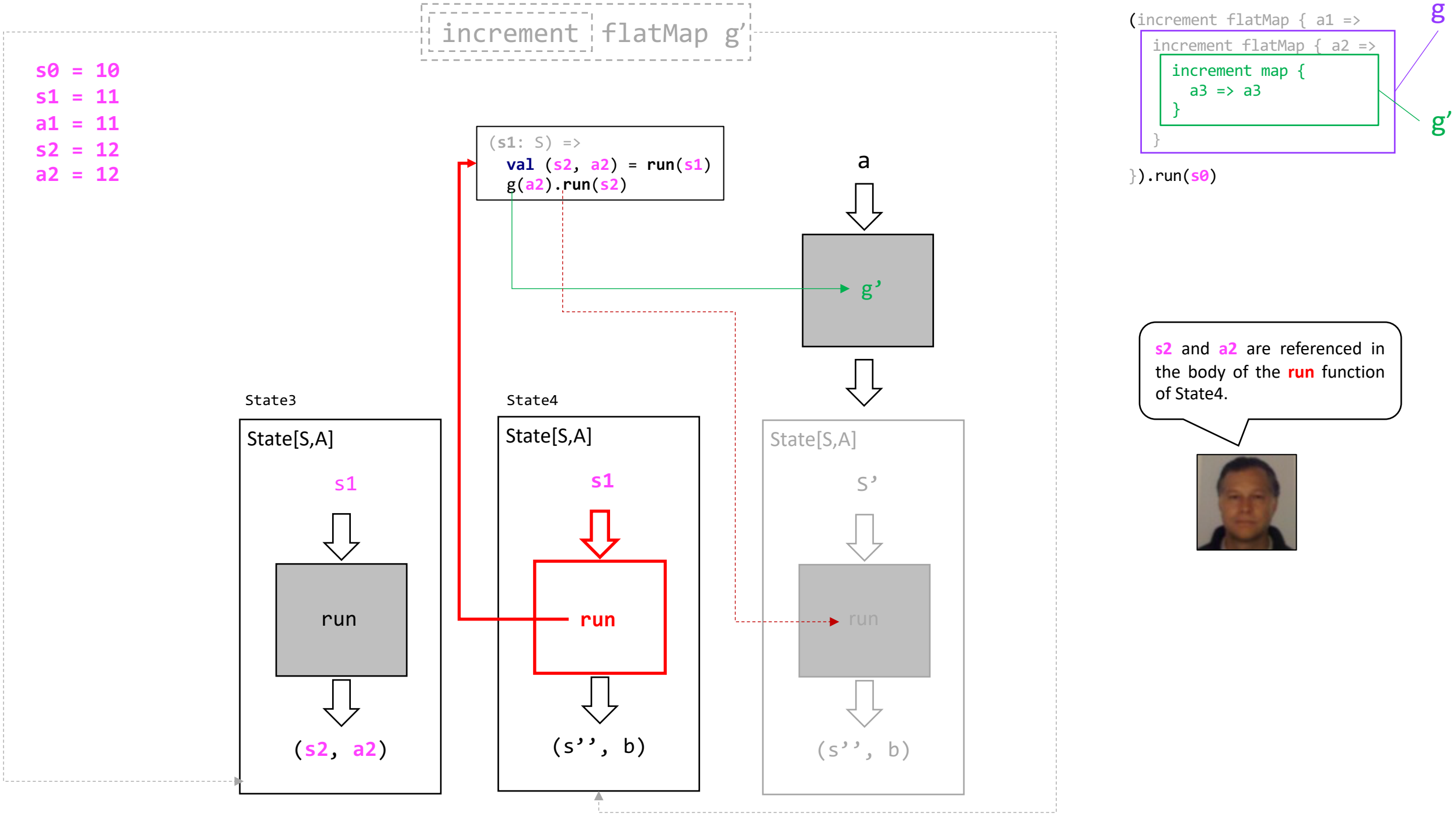
(s’’, b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

The run function of State3 returns new state s2 and value a2.


increment flatMap g’
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
(s1: S) =>
val (s2, a2) = run(s1)
g(a2).run(s2)
a
g’
State3
State[S,A]
s1
run
(s2, a2)
State4
State[S,A]
s1
run
(s’’, b)
State[S,A]
S’
run
(s’’, b)
(increment flatMap { a1 =>
increment flatMap { a2 =>
increment map {
a3 => a3
}
}
}).run(s0)
g
g’
s2 and a2 are referenced in the body of the run function of State4.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

increment flatMap g’

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

f

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

a2

g’

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

The next thing that the **run** function of State4 does is call function **g** with the **a2** value computed by State3.

@philip_schwarz

increment flatMap g’

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

f

g’

State3

State[S,A]

s0

run

(s2, a2)

State4

State[S,A]

s0

run

(s’’, b)

State[S,A]

S’

run

(s’’, b)

The second part of the evaluation of **increment map f**, over the next 8 slides, is a bit different from the evaluation of **increment flatMap g’**, so you probably only want to fast-forward through the next 4 slides.

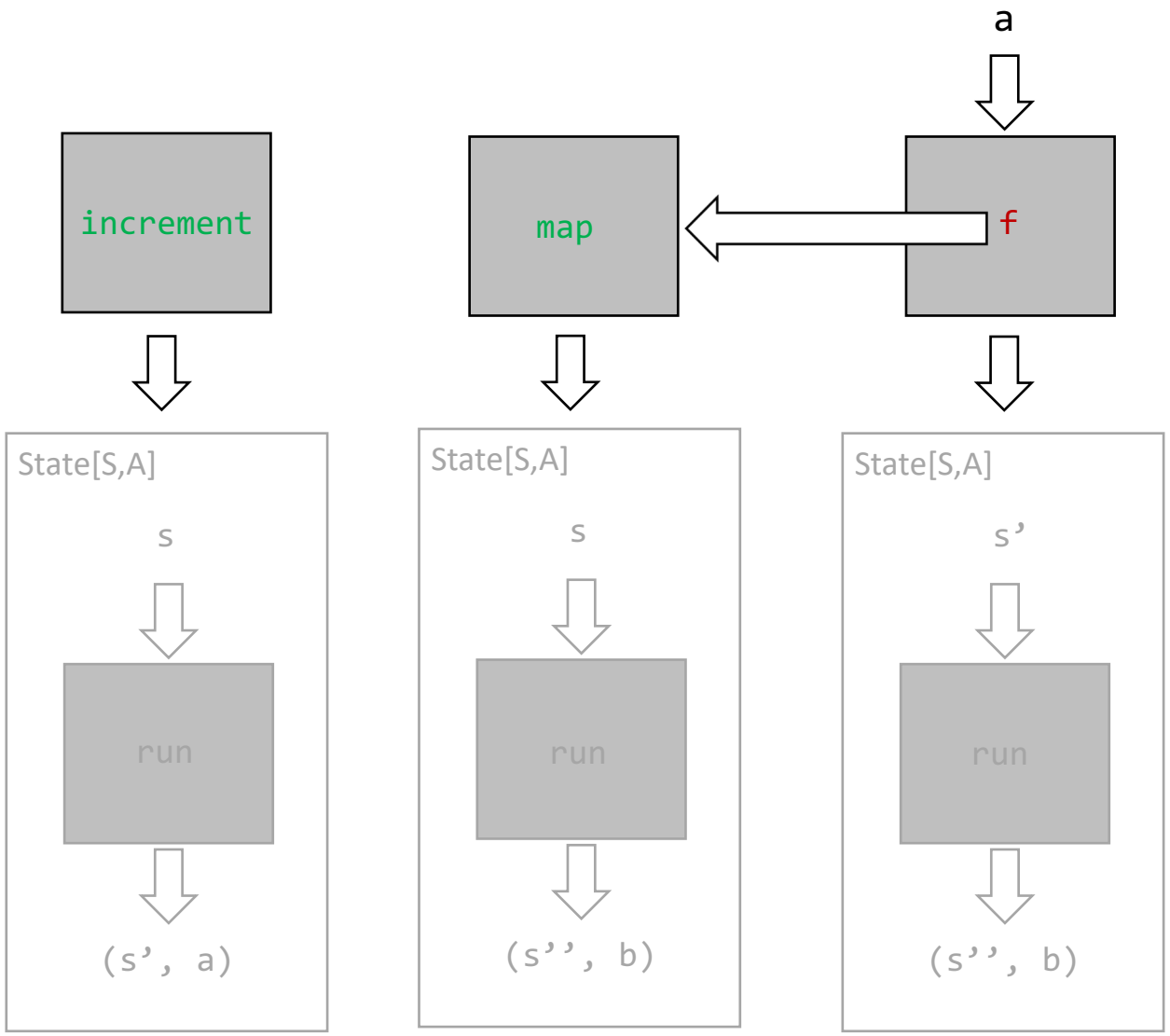
```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

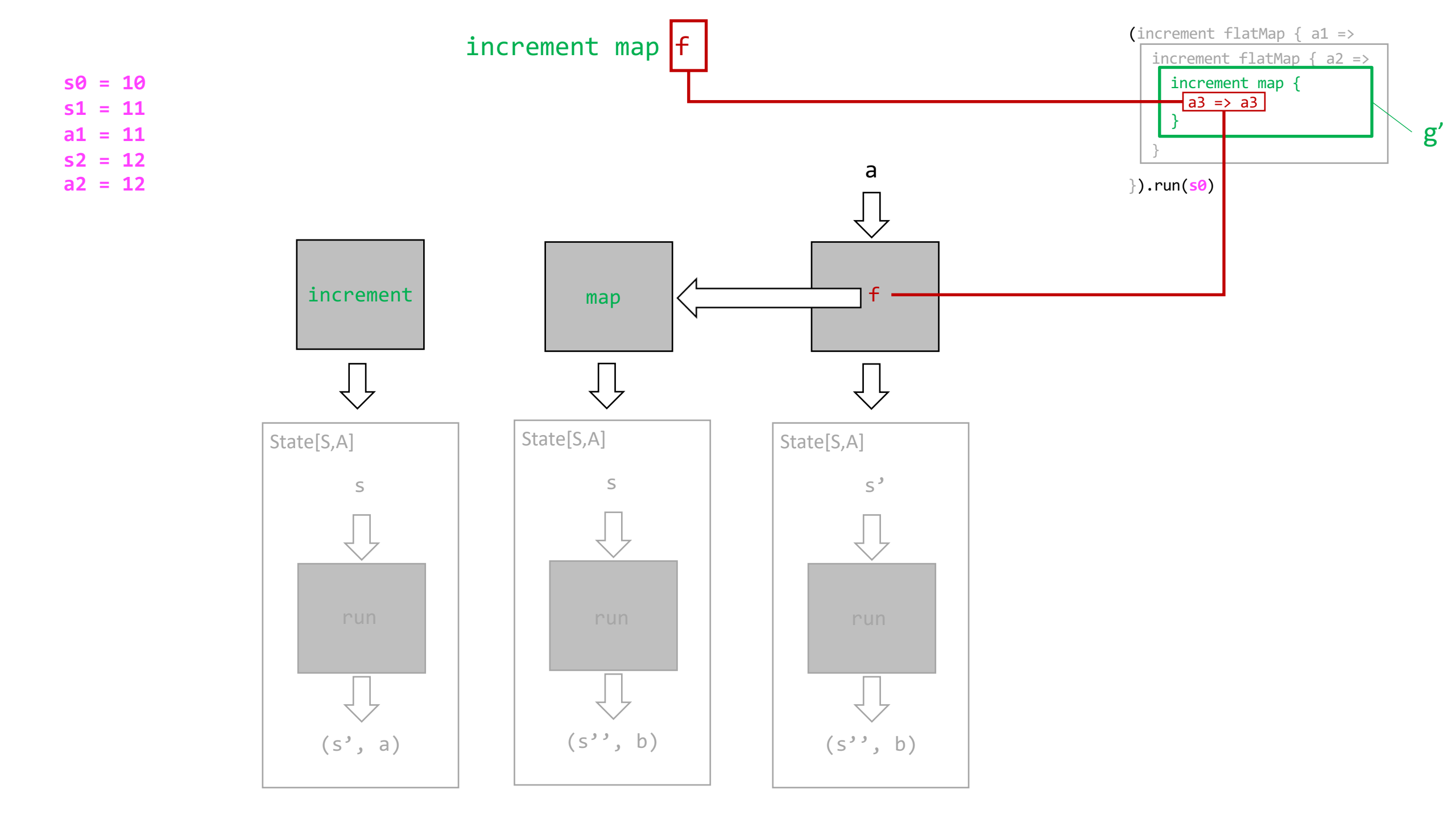
```
increment map f
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

f




s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
increment map f
(increment flatMap { a1 =>
increment flatMap { a2 =>
increment map {
a3 => a3
}
}
}).run(s0)
g'
a
increment
map
f
State[S,A]
s
run
(s', a)
State[S,A]
s
run
(s'', b)
State[S,A]
s'
run
(s'', b)

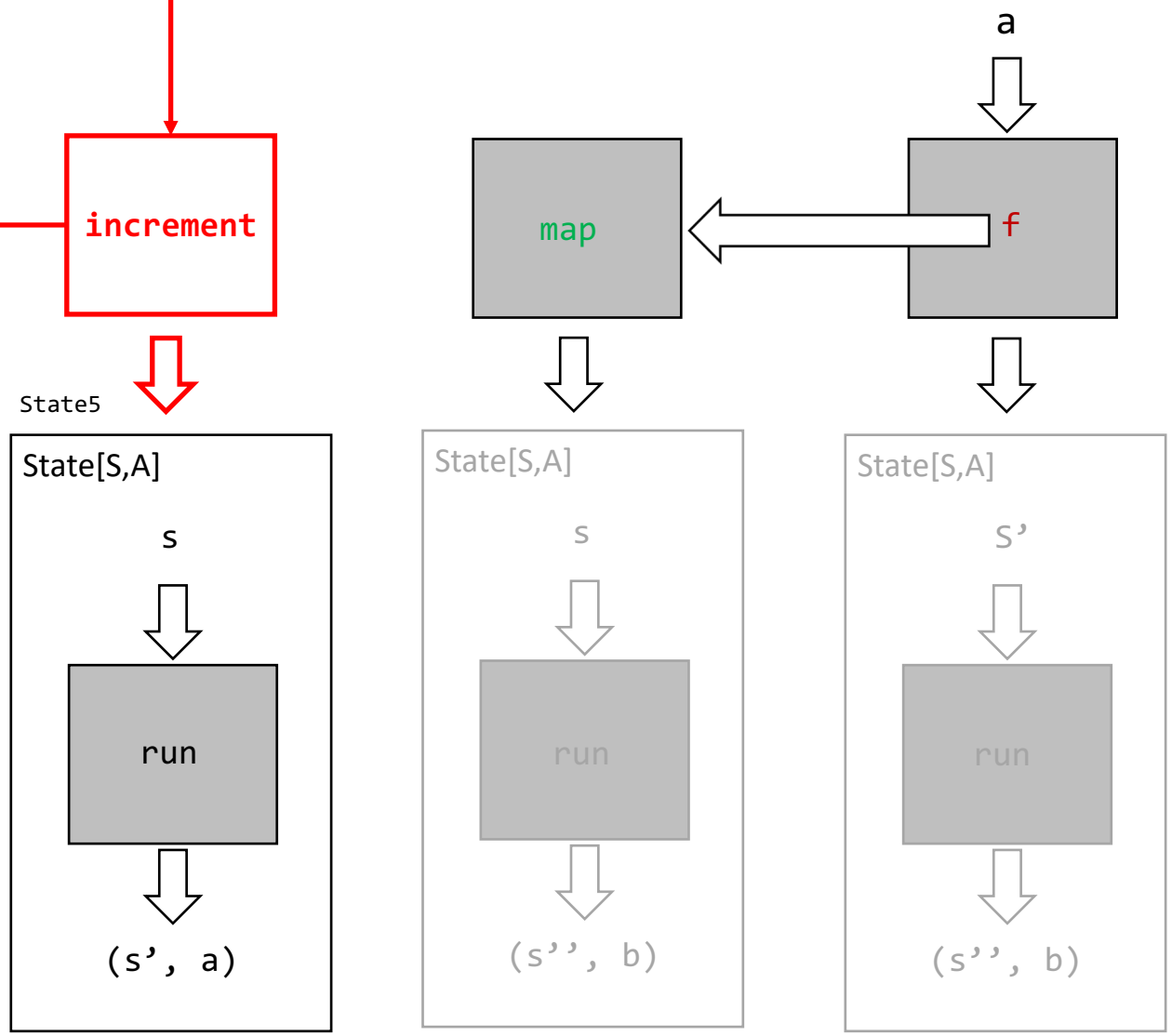
```scala
def increment: State[Int, Int] =
  State { (count: Int) =>
    val nextCount = count + 1
    (nextCount, nextCount)
  }
```

increment map f

```scala
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

f



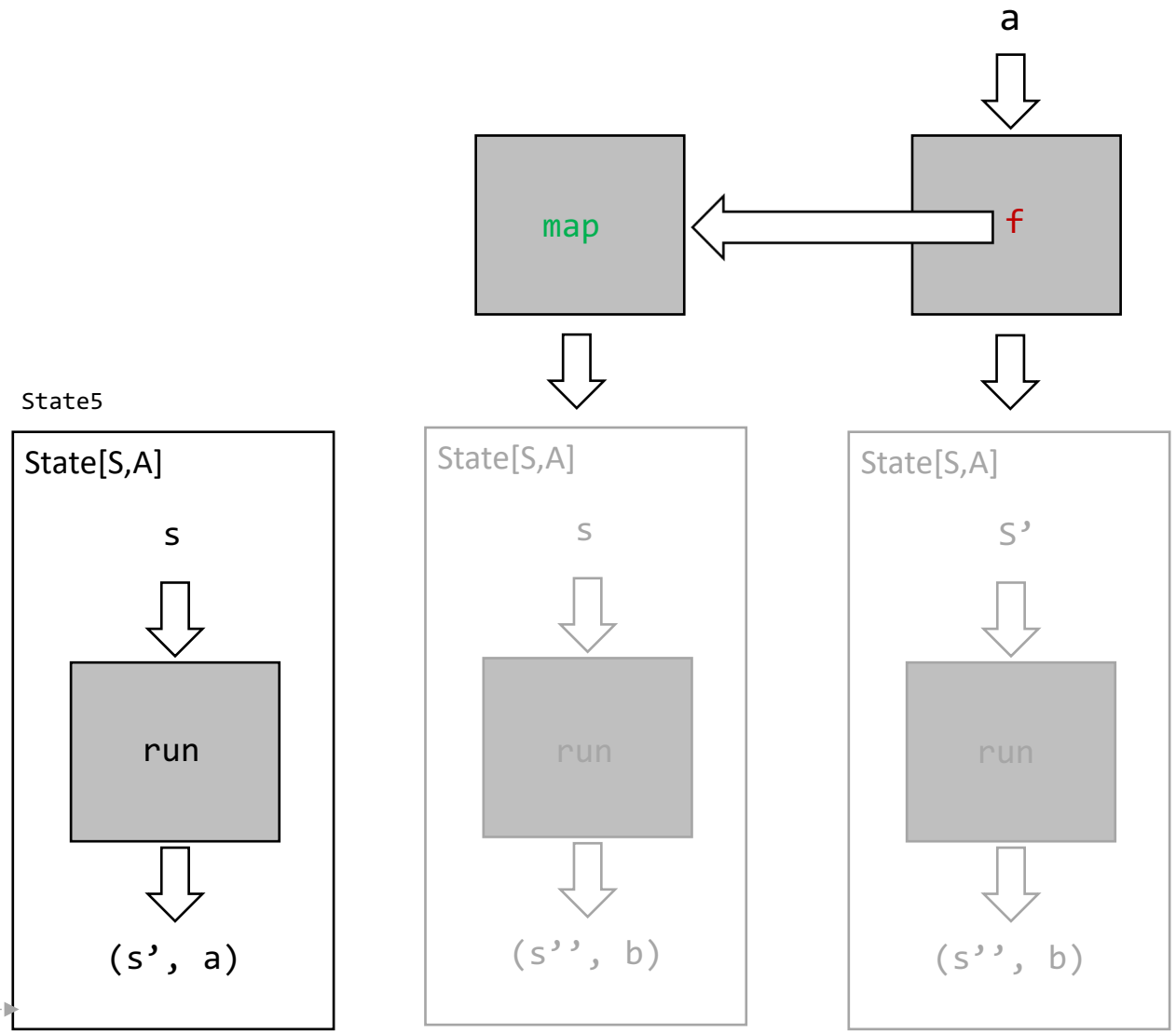
```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

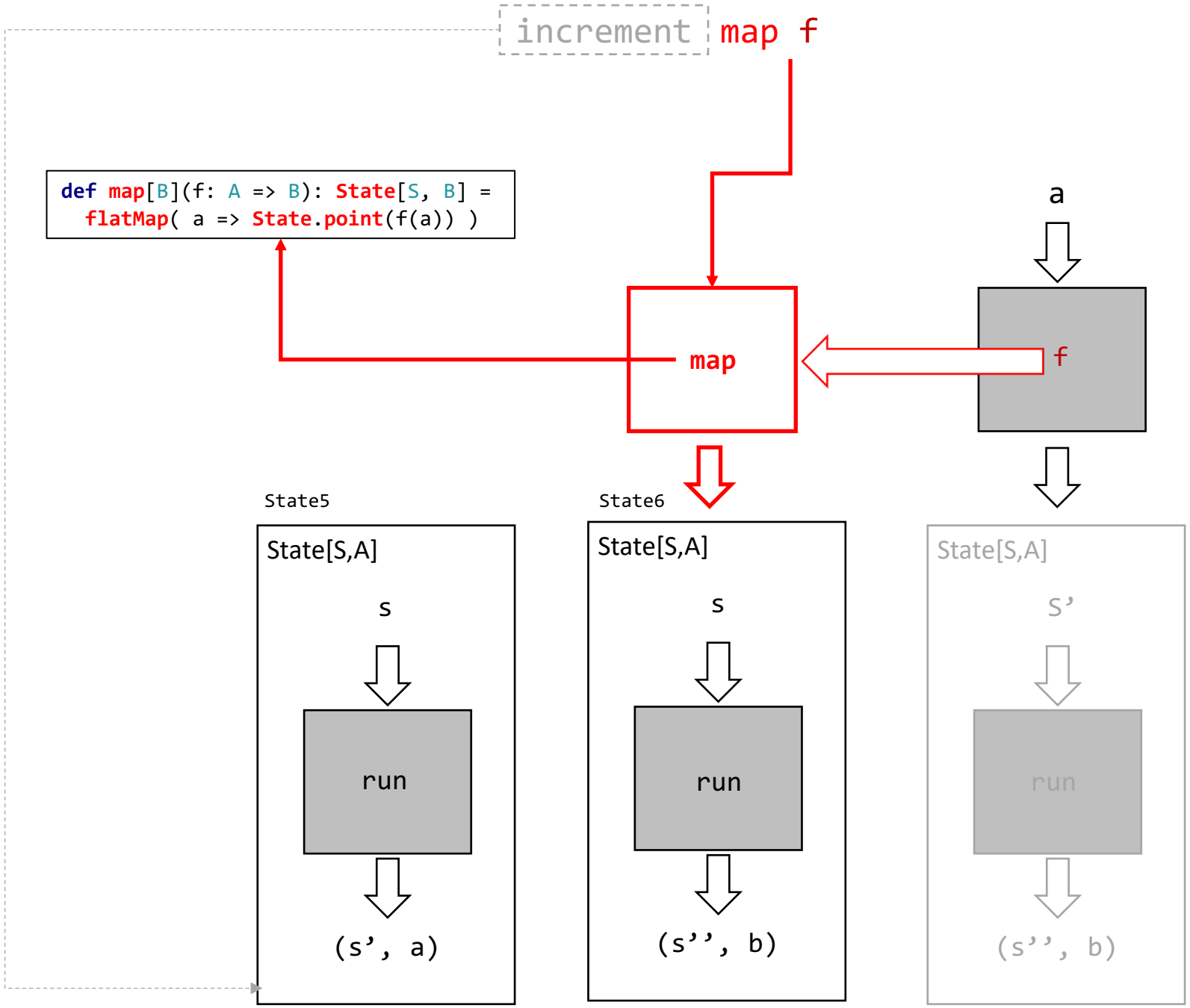
increment map f

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

f






```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

`map(f)` is defined as follows

```
flatMap(a => State.point(f(a)))
```

In this case f is `a3 => a3` i.e. the identity function, so after applying f we are left with

```
flatMap(a => State.point(a))
```

If we then inline the call to **point**, we are left with

```
flatMap(a => State(run = s=>(s,a)))
```

so `map(f)` is `flatMap(g'')` where g'' is

```
a => State( run = s => (s, a) )
```

The next slide is a new version of this slide with the above changes made.

```
def flatMap[B](g: A => State[S, B]): State[S, B] =
  State { (s0: S) =>
    val (s1, a) = run(s0)
    g(a).run(s1)
}
```

increment map f

a => State(run = s => (s, a))

flatMap

a

g''

State5

State[S,A]

s

run

(s', a)

State6

State[S,A]

s

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

Following the points made on the previous slide, instead of invoking the **map** function of State5 with **f**, we invoke its **flatMap** function with **g''** .

We do this because (1) it is similar to what we have been doing so far, which means it is easier to explain and understand (2) it reminds us that **map** is not strictly necessary, i.e. **flatMap** and **point** are sufficient (**map** can be implemented in terms of **flatMap** and **point**).

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

increment map f

```
a => State( run = s => (s, a) )
```

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g''

State5

State[S,A]

s

run

(s', a)

State6

State[S,A]

s

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

We have finished computing **increment map f**. The result is **action state** State6.

In the next slide we return to the execution of the **run** function of State4, which will now make use of State6.

@philip_schwarz

increment flatMap g’

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

We are back to the execution of the **run** function of **state action** State4, at the point where **g** has returned **state action** State6.

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

g’

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g’

f

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

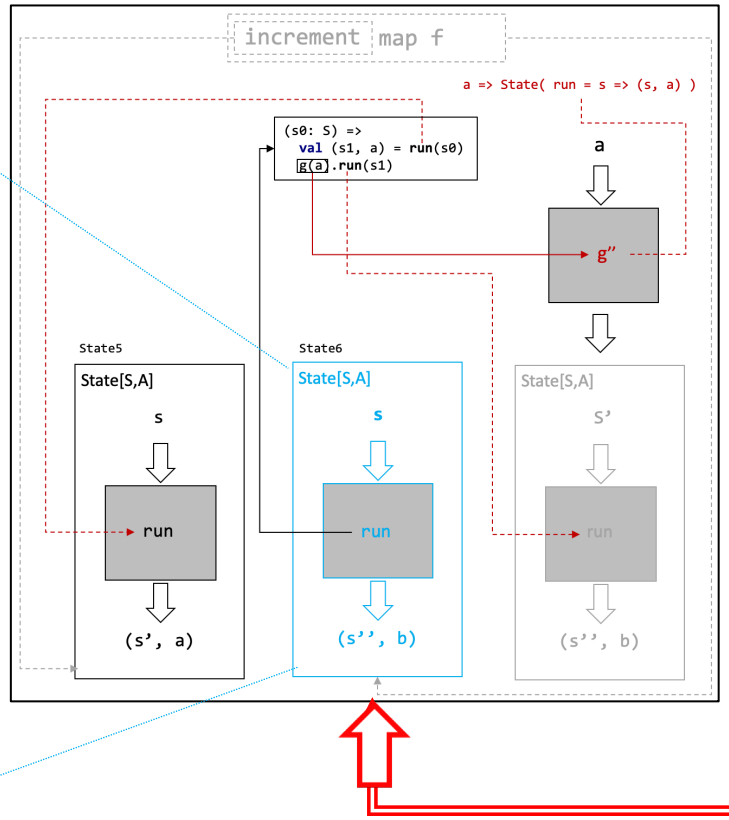
(s’’, b)

State[S,A]

S’

run

(s’’, b)



```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

In the next slide, the **run** function of **state action** State4 is going to invoke the **run** function of **state action** State6.

increment flatMap g'

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

increment map f

a => State(run = s => (s, a))

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g''

State5

State[S,A]

s

run

(s', a)

State6

State[S,A]

s

run

(s'', b)

State[S,A]

s'

run

(s'', b)

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

The **run** function of **state action** State6 is being invoked with the latest **state s2** as a parameter.

Turn to slide 80 if you want to skip to the point where the result of the invocation is returned.

increment flatMap g'

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

increment map f

a => State(run = s => (s, a))

```
(s0: S) =>
  val (s1, a) = run(s0)
  g(a).run(s1)
```

a

g''

State5

State[S,A]

s

run

(s', a)

State6

State[S,A]

s2

run

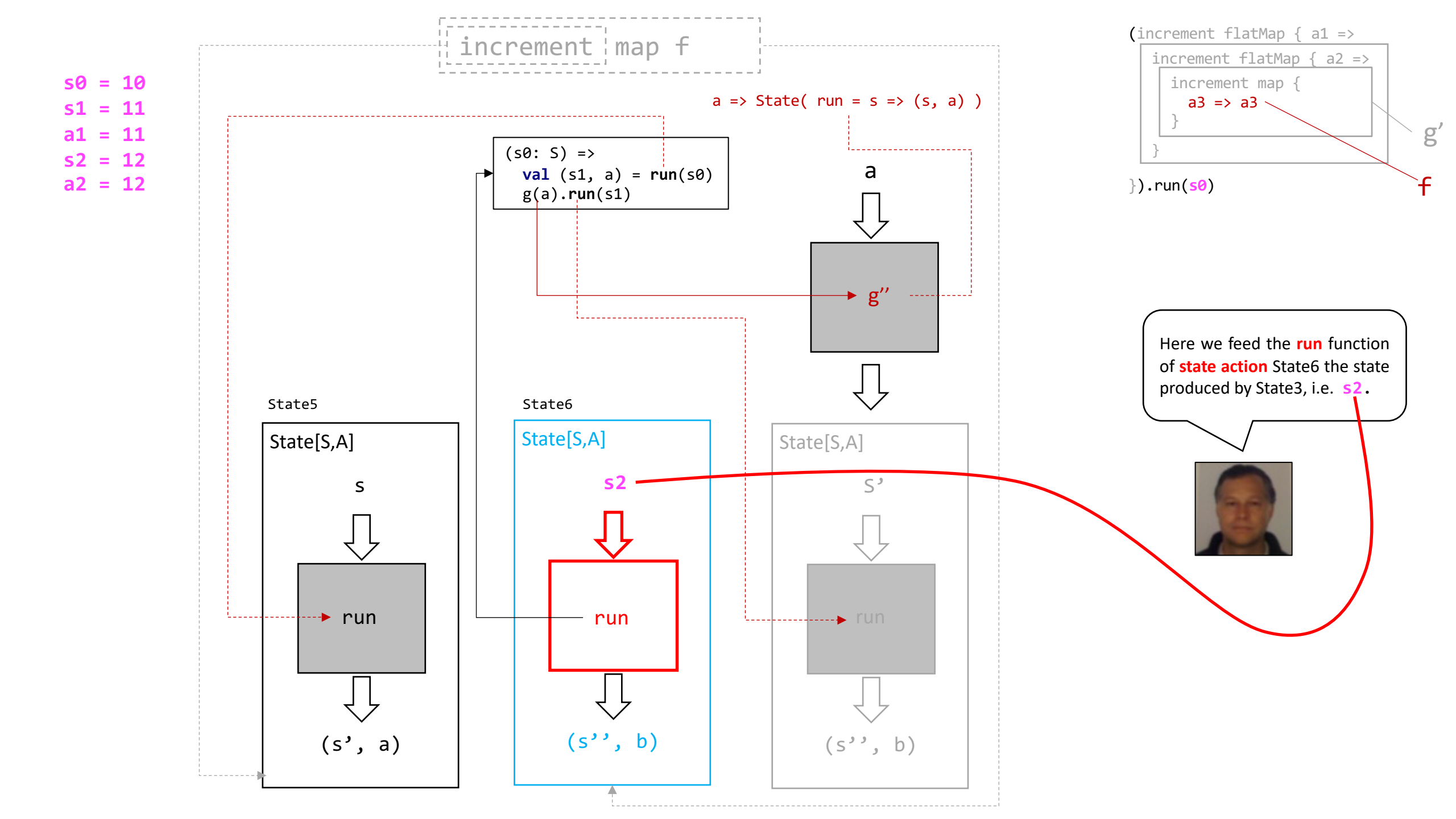
(s'', b)

State[S,A]

s'

run

(s'', b)


s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
increment map f
a => State(run = s => (s, a))
(s0: S) =>
val (s1, a) = run(s0)
g(a).run(s1)
a
g''
State5
State[S,A]
s
run
(s', a)
State6
State[S,A]
s2
run
(s'', b)
State[S,A]
S'
run
(s'', b)
(increment flatMap { a1 =>
increment flatMap { a2 =>
increment map {
a3 => a3
}
}
}).run(s0)
g'
f
Here we feed the run function of state action State6 the state produced by State3, i.e. s2.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
```

increment map f

```
a => State( run = s => (s, a) )
```

```
(s2: S) =>
  val (s3, a3) = run(s2)
  g(a3).run(s3)
```

a

g''

State5

State[S,A]

s

run

(s', a)

State6

State[S,A]

s2

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

Let's pass s2 into the body of State4's run function.

@philip_schwarz

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

increment map f

```
a => State( run = s => (s, a) )
```

```
(s2: S) =>
  val (s3, a3) = run(s2)
  g(a3).run(s3)
```

a

g''

State5

State[S,A]

s2

run

(s3, a3)

State6

State[S,A]

s2

run

(s'', b)

State[S,A]

S'

run

(s'', b)

```
(count: Int) =>
  val nextCount = count + 1
  (nextCount, nextCount)
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

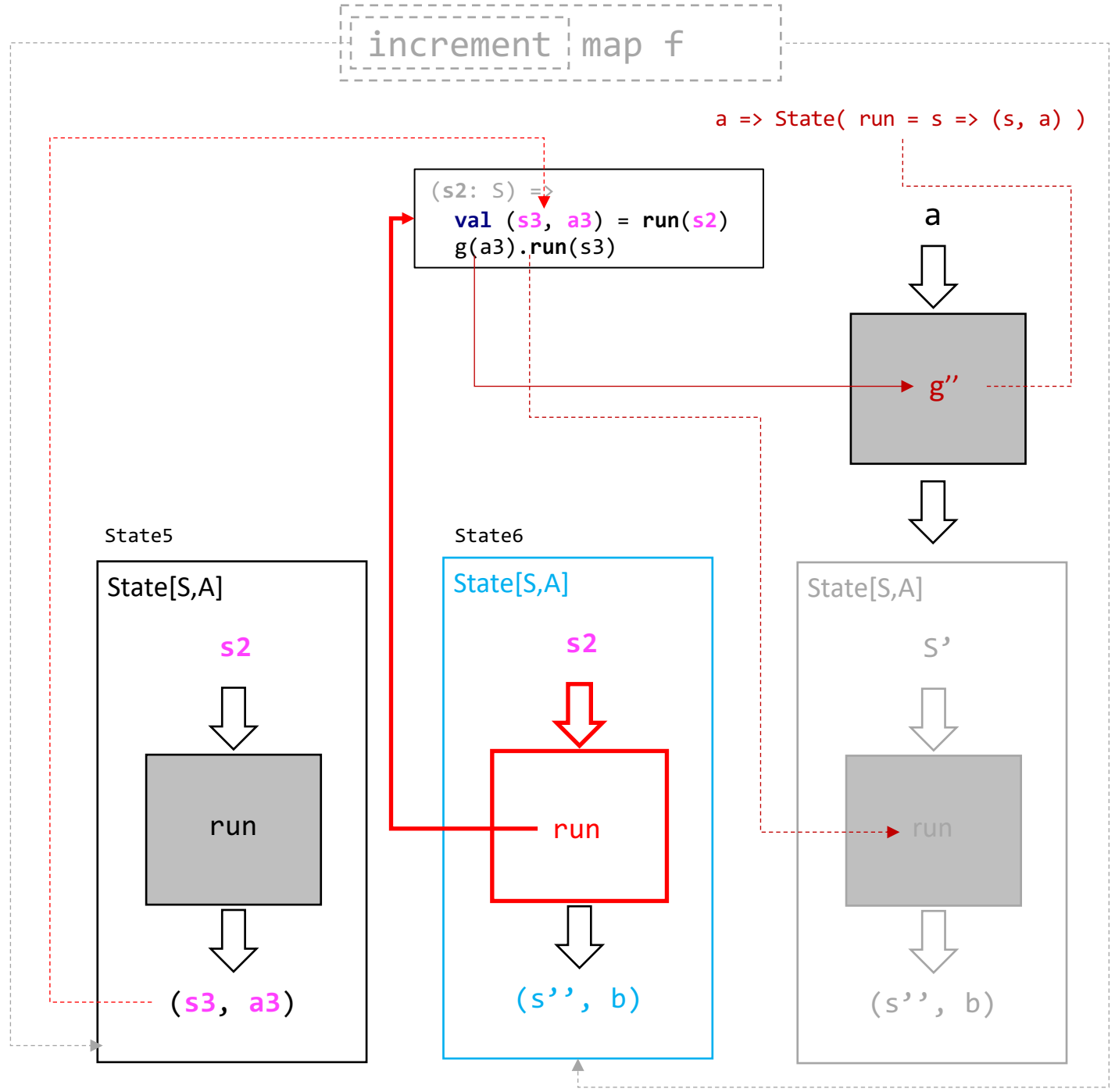
g'

f

The first thing that the **run** function of State6 does is call the **run** function of State5, passing in state **s2**.

This returns (**s3**,**a3**) i.e. the **next state** and a **result** counter, both being **13**.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```



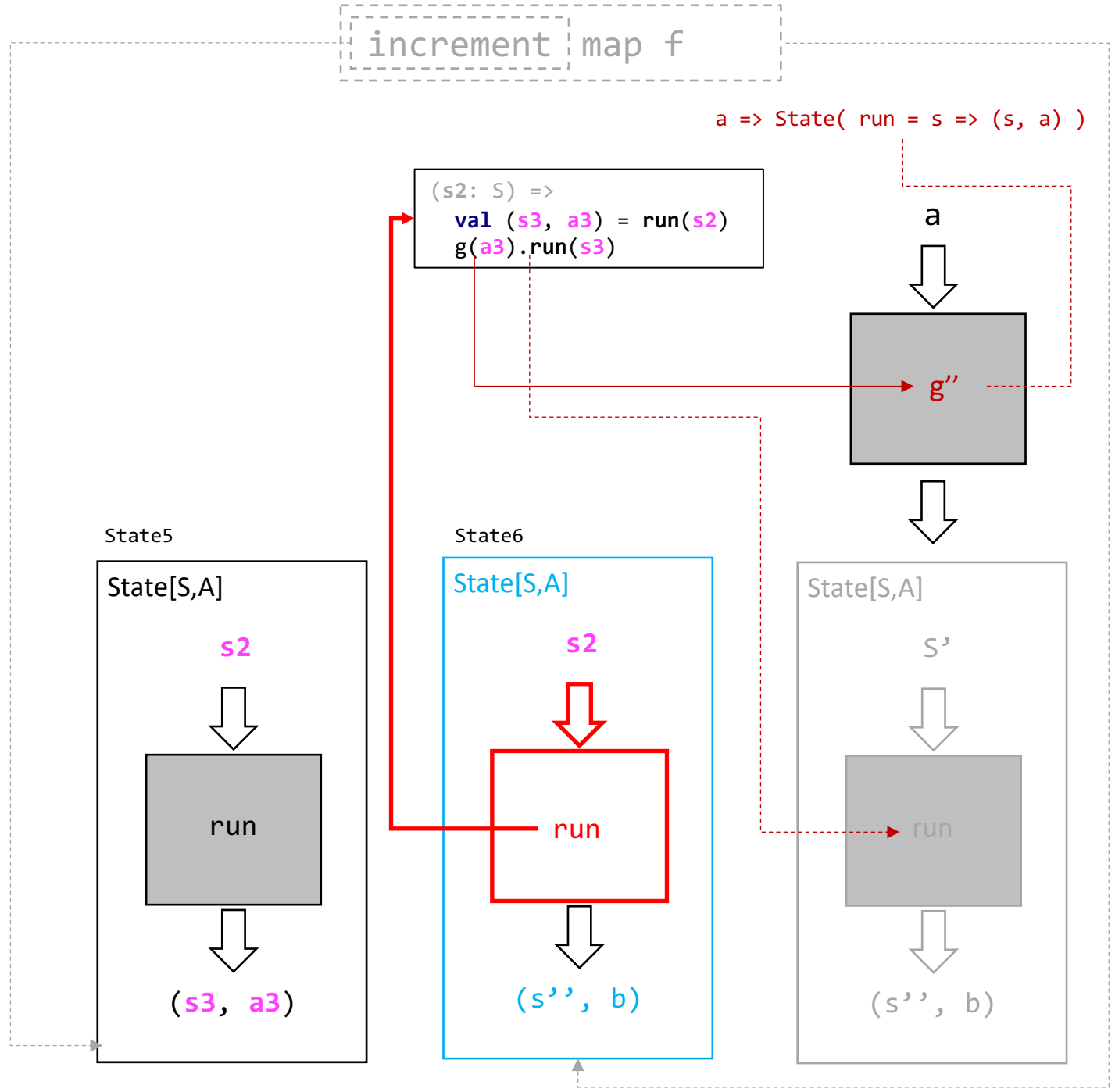

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

The **run** function of State5 returns new state **s3** and value **a3**.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```



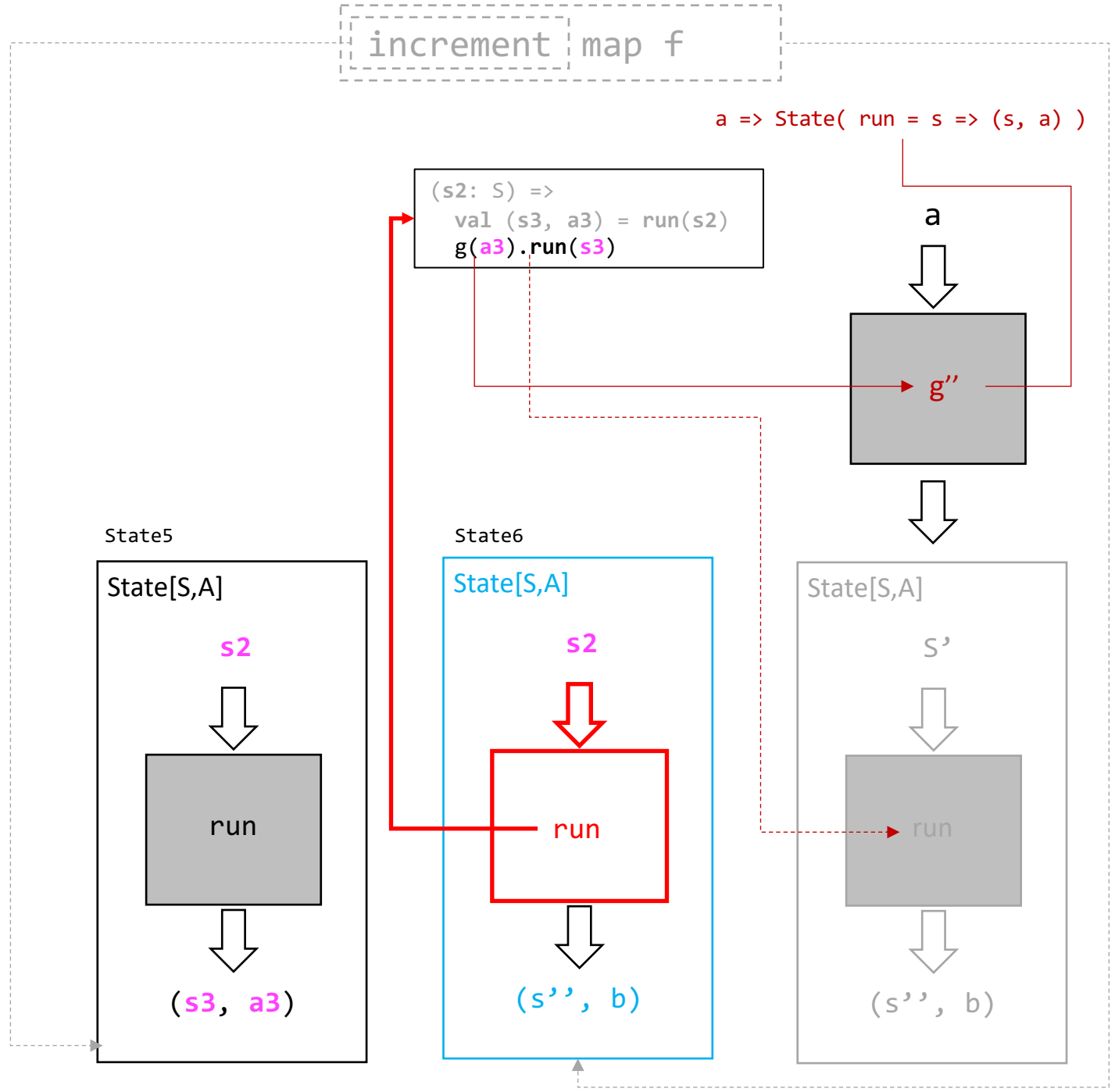

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

s3 and a3 are referenced in the body of the **run** function of State6.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```



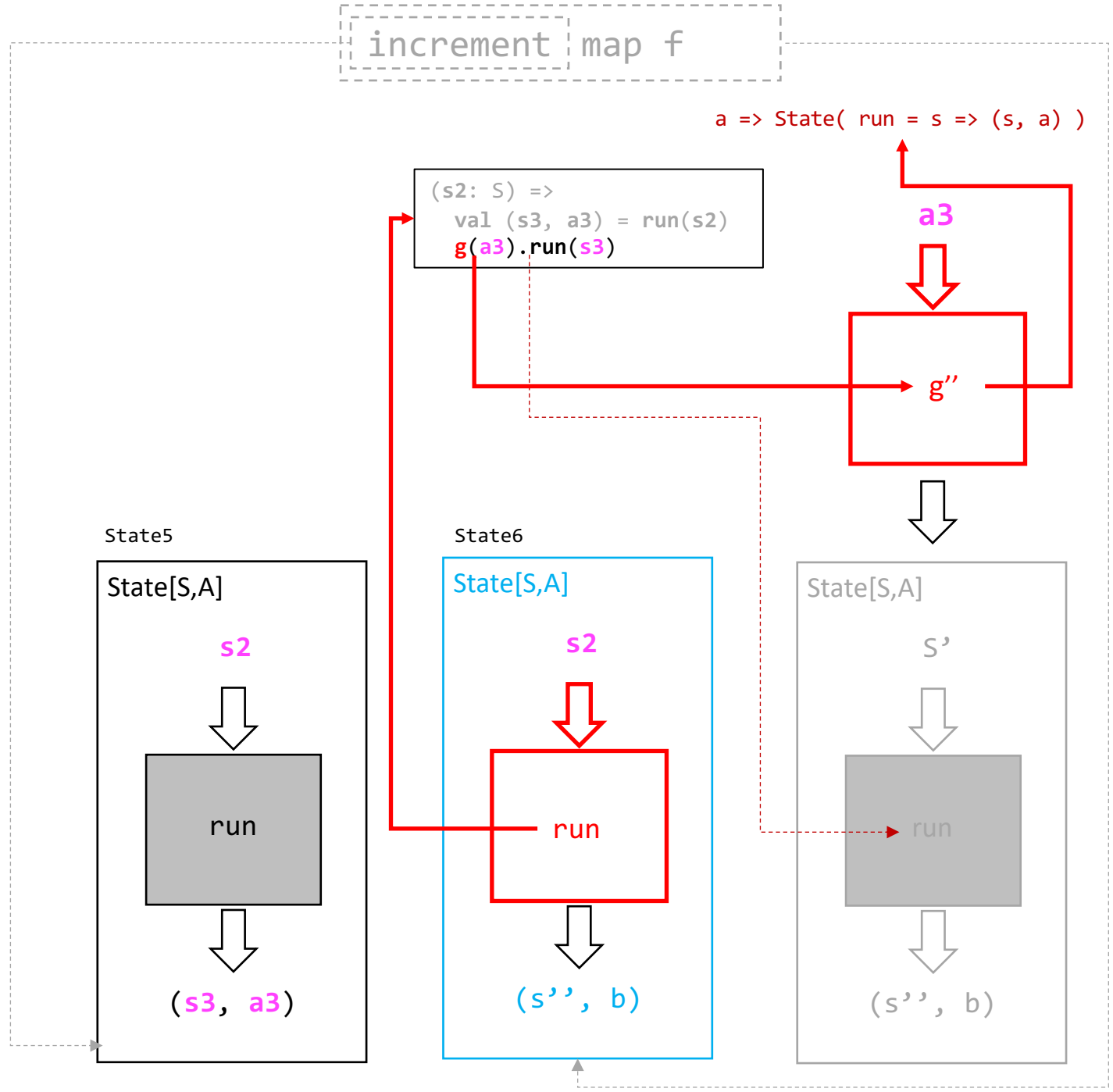

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

The next thing that the **run** function of State6 does is call function **g** with the **a3** value computed by State5.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```




```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

Applying **g''** to **a3** is trivial and produces a new **state action**. See next slide.

@philip_schwarz

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

increment map f

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

s => (s, a)

```
(s2: S) =>
  val (s3, a3) = run(s2)
  g(a3).run(s3)
```

State5

State[S,A]

s2

run

(s3, a3)

State6

State[S,A]

s2

run

(s'', b)

State7

State[S,A]

S'

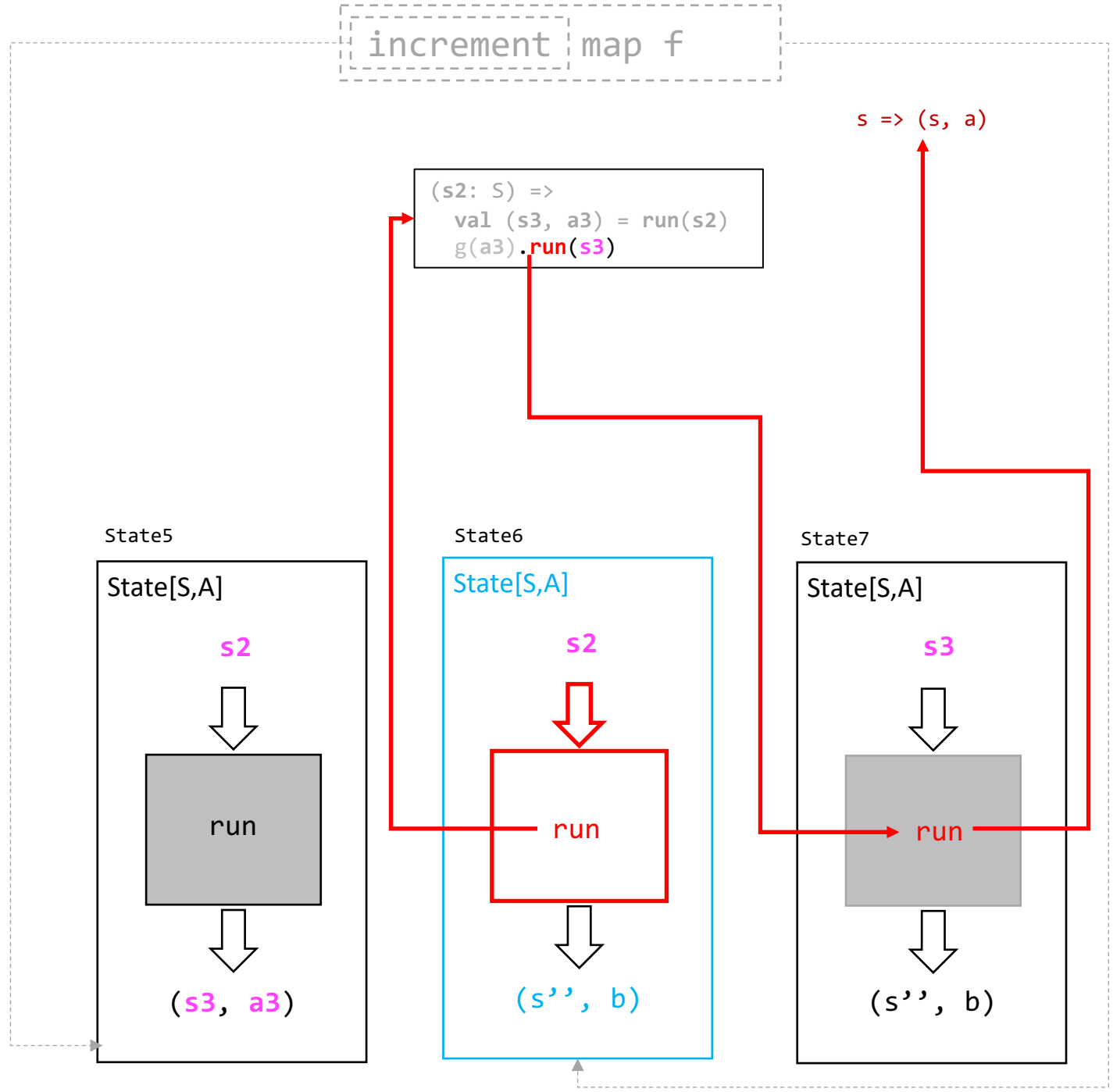
run

(s'', b)

Applying **g''** to **a3** produced new **state action** State7.

In the next slide, the **run** function of State6 is going to invoke the **run** function of State7.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```




```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

The **run** function of State6 invokes the **run** function of State7, passing in the latest state **s3** that was produced by State 5.

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

increment map f

```
(s0: S) =>
  val (s3, a3) = run(s2)
  g(a3).run(s3)
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g'

f

State5

State[S,A]

s2

run

(s3, a3)

State6

State[S,A]

s2

run

(s3, a3)

State7

State[S,A]

s3

run

(s3, a3)

We have finished executing the **run** function of **action state** State7. The result is (s3,a3) , i.e. (13,13).

That is also the result of executing the **run** function of **state action** State6.

In the next slide we return to the execution of the **run** method of State4, which will now make use of result (s3,a3).

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

We have finished executing the **run** function of **action state** State6. The result is (s3,a3) , i.e. (13,13).

increment flatMap g'

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

State3

State[S,A]

s1

run

(s2, a2)

State4

State[S,A]

s1

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

increment map f

```
(s0: S) =>
  val (s3, a3) = run(s2)
  g(a3).run(s3)
```

State5

State[S,A]

s2

run

(s3, a3)

State6

State[S,A]

s2

run

(s3, a3)

State[S,A]

s3

run

(s3, a3)

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

The result of the **run** function of **action state** State6 is also the result of the **run** function of **action state** State4: see next slide.

@philip_schwarz

increment flatMap g'

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

State3

State[S,A]

s0

run

(s1, a1)

State4

State[S,A]

s1

run

(s'', b)

State6

State[S,A]

s2

run

(s3, a3)

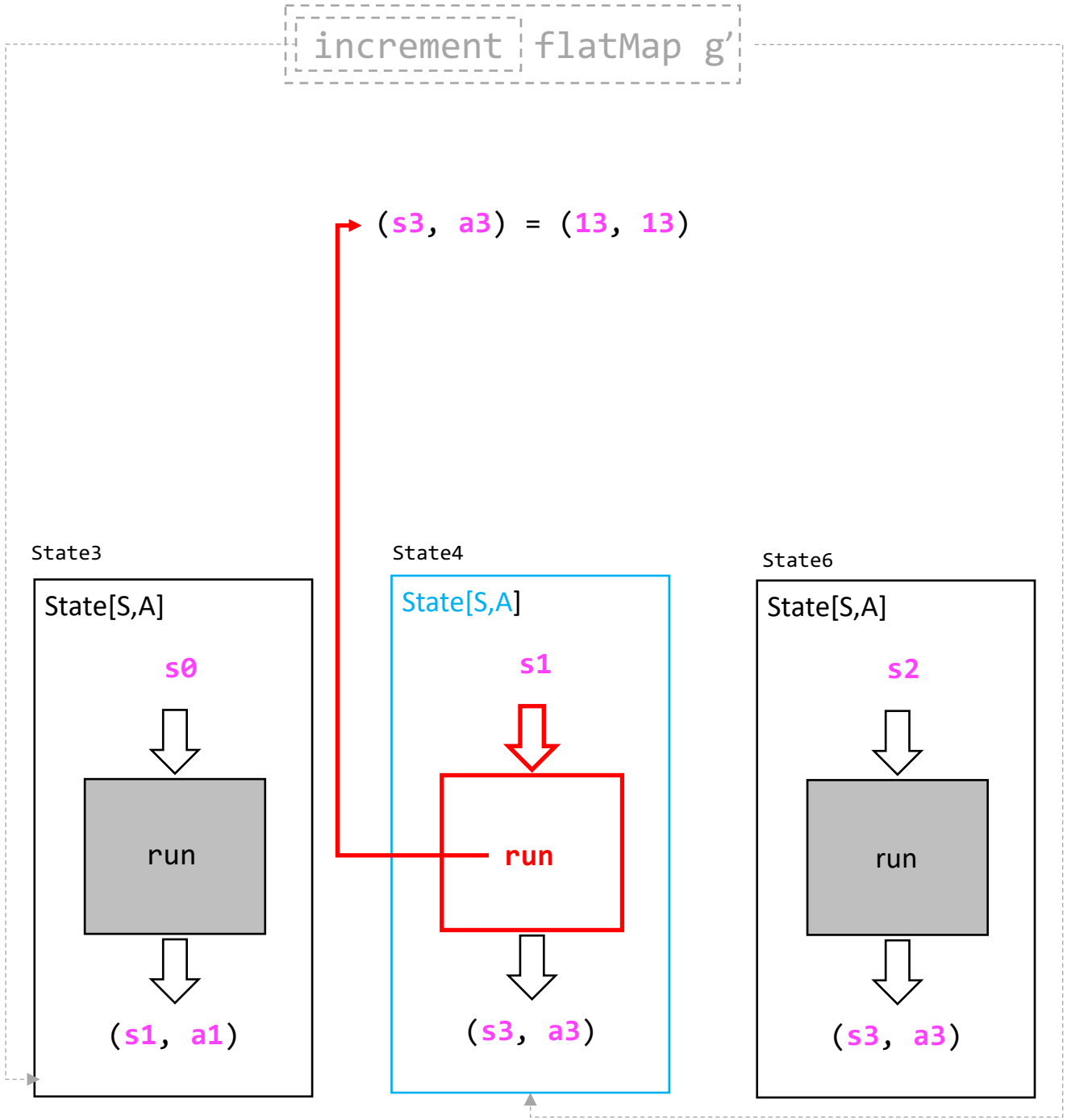
```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```




```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

The result of the **run** function of **action state** State4 is (s3,a3) , i.e. (13,13).

In the next slide we return to the execution of the **run** method of State2, which will now make use of result (s3,a3).

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

We have finished executing the **run** function of **action state** State4. The result is (s3,a3) , i.e. (13,13).

increment flatMap g

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

State1

State[S,A]

s0

run

(s1, a)

State2

State[S,A]

s0

run

(s'', b)

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

increment flatMap g'

```
(s1: S) =>
  val (s2, a2) = run(s1)
  g(a2).run(s2)
```

State3

State[S,A]

s0

run

(s1, a1)

State4

State[S,A]

s1

run

(s3, a3)

State6

State[S,A]

s2

run

(s3, a3)

increment flatMap g

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

The result of the **run** function of **action state** State4 is also the result of the **run** function of **action state** State2: see next slide.

```
(s0: S) =>
  val (s1, a1) = run(s0)
  g(a1).run(s1)
```

State1

State[S,A]

s0

run

(s1, a1)

State2

State[S,A]

s0

run

(s’’, b)

State4

State[S,A]

s1

run

(s3, a3)

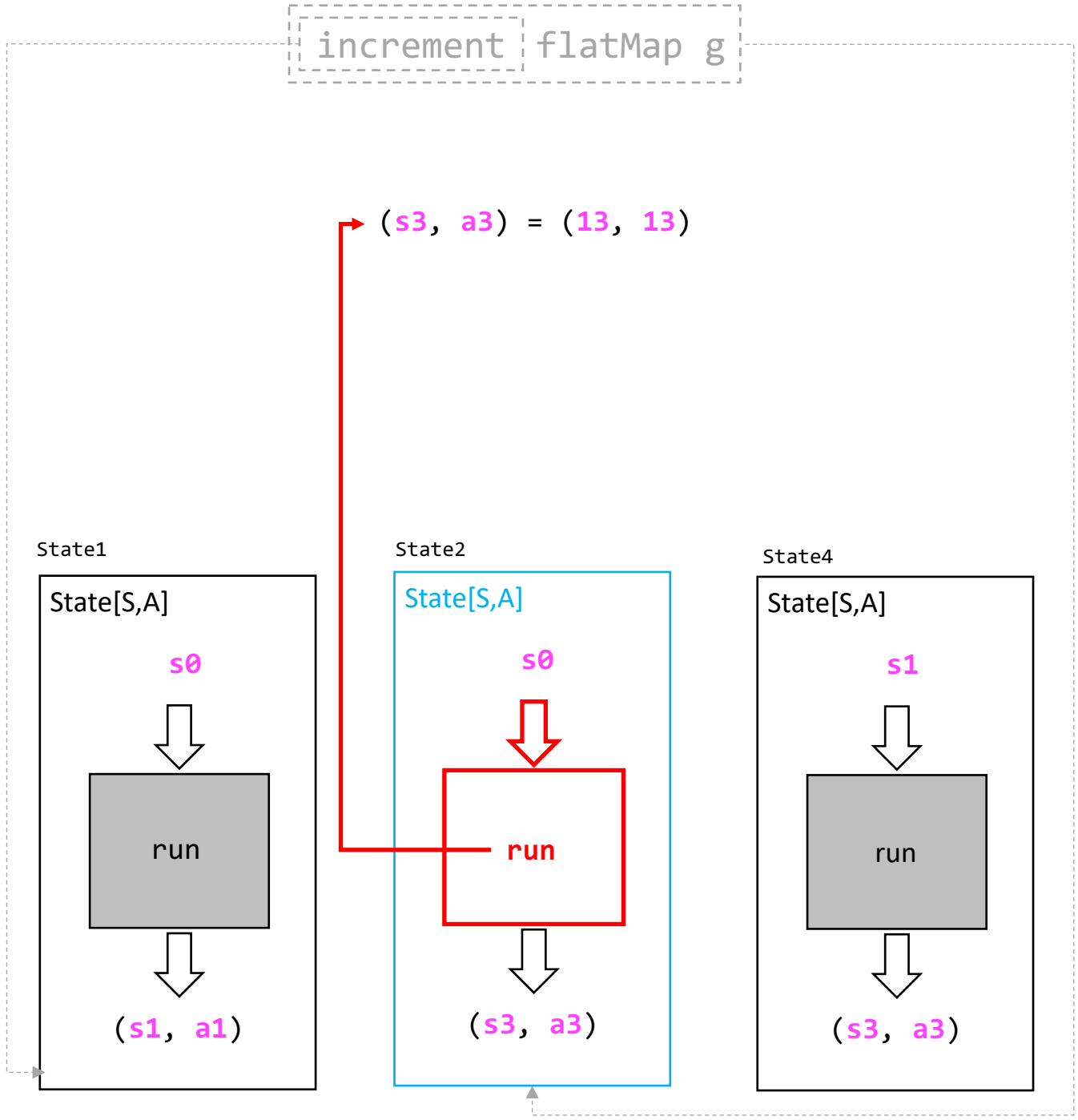
```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g’

f

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```




```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

The result of the **run** function of **action state** State2 is (s3,a3) , i.e. (13,13).

@philip_schwarz

```
s0 = 10
s1 = 11
a1 = 11
s2 = 12
a2 = 12
s3 = 13
a3 = 13
```

```
(increment flatMap g).run(s0)) == (s3,a3)
                    10             (13,13)
```

```
(s3, a3) = (13, 13)
```

```
increment flatMap { a1 =>
    increment flatMap { a2 =>
      increment map { a3 =>
        a3
      }
    }
  }.run(10)
```

```
(increment flatMap { a1 =>
  increment flatMap { a2 =>
    increment map {
      a3 => a3
    }
  }
}).run(s0)
```

g

g'

f

State1

State[S,A]

S0 = 10

run

(s1, a1) =
(11, 11)

State2

State[S,A]

s0 = 10

run

(s3, a3) =
(13, 13)

State4

State[S,A]

s1 = 11

run

(s3, a3) =
(13, 13)

This is also the result of the whole computation that started on slide 29 with the evaluation of **increment flatMap g** and continued on slide 34 with the execution, with an **initial count state** of **10**, of the **run** function of the resulting **composite action state**.

Let's not forget that in this simplest of examples, in which we use the **State monad** simply to model a counter, **none of the intermediate counter values computed by the state actions are put to any use, they are just ignored**. The only thing we are interested in is the computation of the sequence of states leading to the value produced by the final state transition.

```
def increment: State[Int, Int] =
  State { (count: Int) =>
    val nextCount = count + 1
    (nextCount, nextCount)
  }

val tripleIncrement: State[Int, Int] =
  for {
    _         <- increment
    _         <- increment
    result <- increment
  } yield result

val initialState = 10

val (finalState, result) =
  tripleIncrement.run(initialState)

assert( finalState == 13 )
assert( result == 13 )
```

In the next slide deck in this series we'll look at examples where the intermediate values do get used.

I also want to stress again the point that when we use **map** and **flatMap** to **compose state actions**, **no state transitions occur, no results are computed**.

A **composite state action** is produced, but **until its run function is invoked with an initial state, nothing happens**.

It is up to us to decide if and when to invoke the run function. **It is when we invoke the run function that the state transitions occur** (all of the state transitions) **and results are computed** (all of the results, intermediate ones and final one), **all in one go, triggered by our invocation of the run function**. This is unlike some other monads, e.g. **Option**.

To illustrate this further, see the example on the next slide.

```scala
val maybeCompositeMessage =
  Some("Hello") flatMap { greeting =>
    println("in function passed to 1st flatMap")
    Some("Fred") flatMap { name =>
      println("in function passed to 2nd flatMap")
      Some("Smith") map { surname =>
        println("in function passed to map")
        s"$greeting $name $surname!"
      }
    }
  }
```

```scala
val compositeStateAction =
  increment flatMap { a1 =>
    println("in function passed to 1st flatMap")
    increment flatMap { a2 =>
      println("in function passed to 2nd flatMap")
      increment map { a3 =>
        println("in function passed to map")
        a1 + a2 + a3
      }
    }
  }
```

@philip_schwarz

In the case of computing the **composite Option**, the functions passed to **map** and **flatMap** are all exercised during that computation and the result we seek is contained in the resulting option.

In the case of computing the **composite state action**, none of the functions passed to **map** and **flatMap** are exercised during that computation. The result we seek is obtained, at a later time of our choosing, from the resulting **composite state action**, by invoking its **run** function with an **initial state**, and it is only at that point that the functions passed to **map** and **flatMap** are exercised.

```
scala> val maybeCompositeMessage =
     |   Some("Hello") flatMap { greeting =>
     |   println("in function passed to 1st flatMap")
     |     Some("Fred") flatMap { name =>
     |     println("in function passed to 2nd flatMap")
     |       Some("Smith") map { surname =>
     |       println("in function passed to map")
     |         s"$greeting $name $surname!"
     |       }
     |     }
     |   }
in function passed to 1st flatMap
in function passed to 2nd flatMap
in function passed to map
maybeCompositeMessage: Option[String] = Some(Hello Fred Smith!)
```

```
scala> val compositeStateAction =
     |   increment flatMap{ a1 =>
     |     println("in function passed to 1st flatMap")
     |     increment flatMap { a2 =>
     |       println("in function passed to 2nd flatMap")
     |       increment map { a3 =>
     |         println("in function passed to map")
     |         a1 + a2 + a3
     |       }
     |     }
     |   }
compositeStateAction: State[Int,Int] =
State(State$$Lambda$1399/2090142523@7d6ccad7)

scala> compositeStateAction.run(2)
in function passed to 1st flatMap
in function passed to 2nd flatMap
in function passed to map
res0: (Int, Int) = (5,12)
```

In this last slide, let me have a go at describing in words, in an informal way that you may or may not find useful, what the **flatMap** function of the **State monad** does.

```
case class State[S, A](run: S => (S, A)) {

  def flatMap[B](g: A => State[S, B]): State[S, B] =
    State { (s0: S) =>
      val (s1, a) = run(s0)
      g(a).run(s1)
    }

  def map[B](f: A => B): State[S, B] =
    flatMap( a => State.point(f(a)) )
}

object State {
  def point[S, A](v: A): State[S, A] = State(run = s => (s, v))
}
```

I am not bothering with a separate description of the **map** function since it can be understood in terms of **flatMap** and **point**.

Given a **state action** `sa1:State[S,A]` and a callback function **g** that represents the rest of the program and which takes an A and returns a **state action** `sa2:State[S,B]`, **flatMap** returns a new, **composite state action** `sa3:State[S,B]` whose **run** function, when invoked with state **s0**, does the following:

1) first invokes the **run** function of **sa1** with **s0**, producing next state **s1** and result **a**.
2) then invokes callback function **g**, the rest of the program, with result **a**, producing new **state action sa3**.
3) finally invokes the **run** function of **sa3** with intermediate state **s1**, producing next state **s2** and result **b**.

To be continued in part II