Scala.Rx

Scaladays 2014, Berlin

Li Haoyi

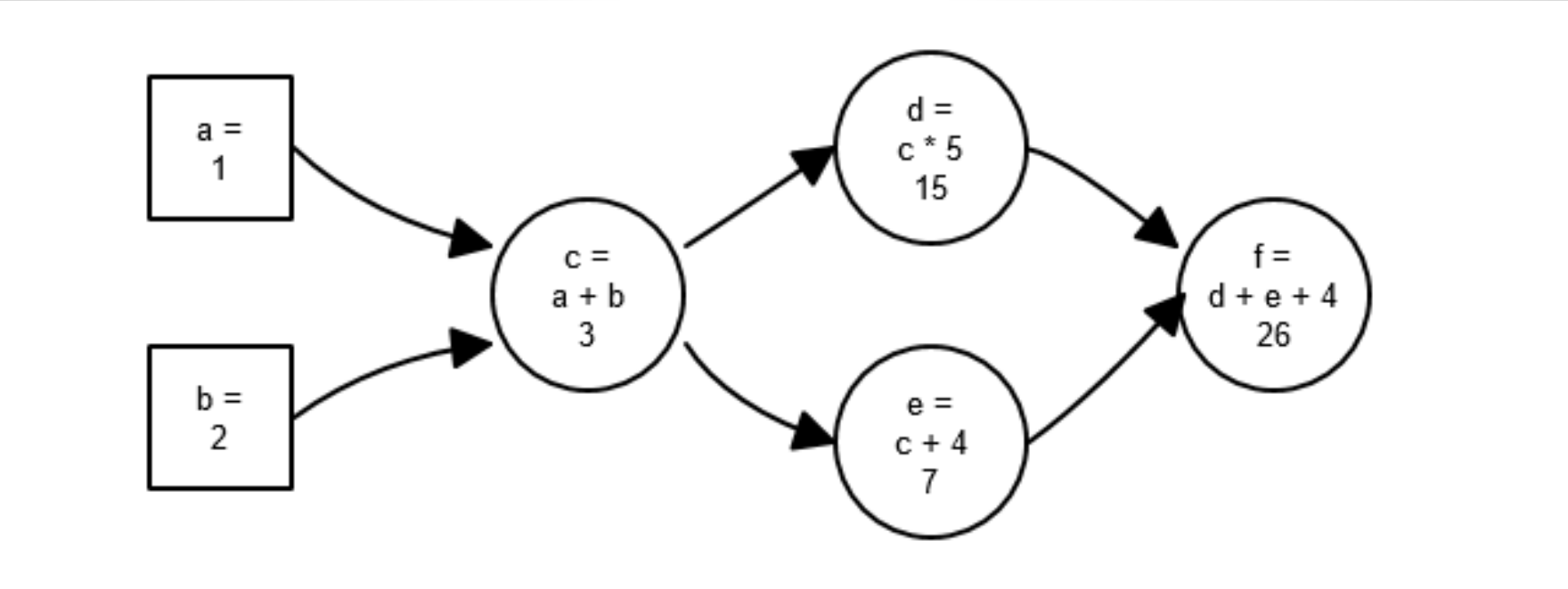
https://github.com/lihaoyi/scala.rx

What

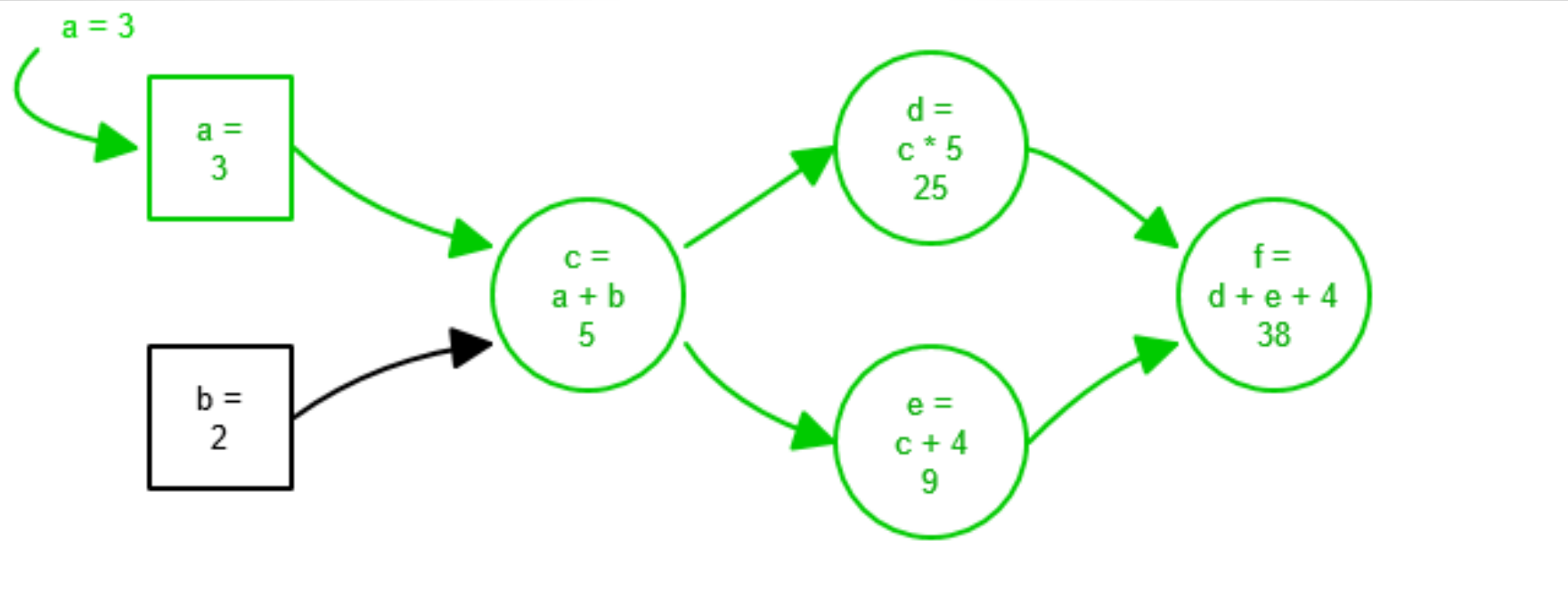
- `libraryDependencies += "com.scalarx" %% "scalarx" % "0.2.5"`

- Scala.Rx is a change-propagation library
- Reactive values which depend on each other
- Change one and they propagate the update

Reactive values which depend on each other



Change one and they propagate the update



Motivation

```
var a = 1; var b = 2
val c = a + b
println(c) // 3
a = 4
println(c) // 3
```

Motivation

```
var a = 1; var b = 2

def c = a + b

println(c) // 3

a = 4

println(c) // 6
```

Motivation

```
var a = 1; var b = 2
def c = veryExpensiveOperation(a, b)
println(c) // 3
a = 4
println(c) // 6
```

Motivation

```
var a = 1; var b = 2

def c = a + b

// onChange(c, () => ...)

a = 4
```

Motivation

```
import rx._
val a = Var(1); val b = Var(2)
val c = Rx{ a() + b() }
println(c()) // 3
a() = 4
println(c()) // 6
```

Motivation

```
import rx._
val a = Var(1); val b = Var(2)
val c = Rx{ a() + b() }
println(c()) // 3
a() = 4
println(c()) // 6
Obs(c){ ... do something... }
```

What

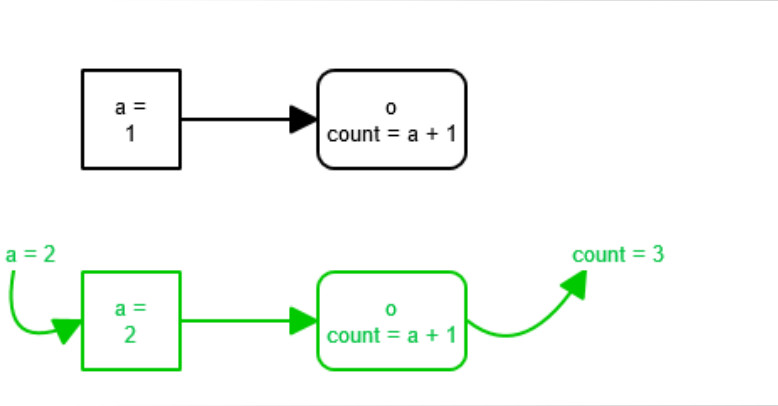
- `Var`: reactive variables that are set manually
- `Rx`: reactive values that depend on other reactive values
- `Obs`: observes changes to reactive values and does things

Why

- Most mutable state isn’t really “state”
  - Depends on other variables
  - Should be kept in sync
  - Weird things happen if it falls out of sync?
- When recalculating something, you want to do it the same way you did it the first time
- Scala.Rx saves you from having to keep things in sync manually

What - Observers

```
val a = Var(1)
var count = 0
val o = Obs(a){
  count = a() + 1
}
println(count) // 2
a() = 4
println(count) // 5
```



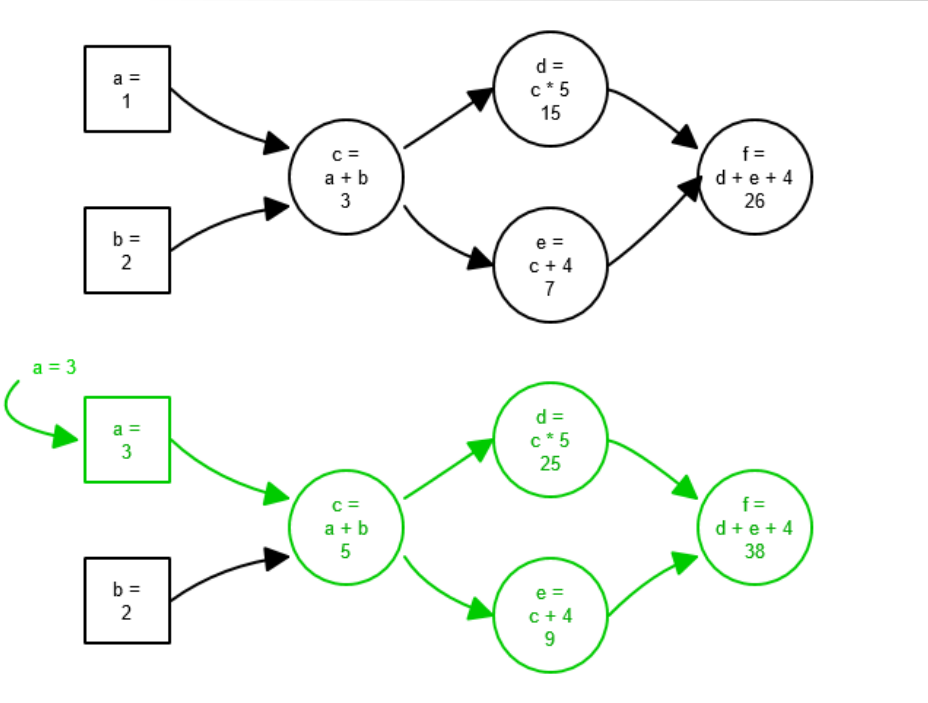
What - Propagation

```
val a = Var(1) // 1

val b = Var(2) // 2

val c = Rx{ a() + b() } // 3
val d = Rx{ c() * 5 } // 15
val e = Rx{ c() + 4 } // 7
val f = Rx{ d() + e() + 4 } // 26

println(f()) // 26
a() = 3
println(f()) // 38
```

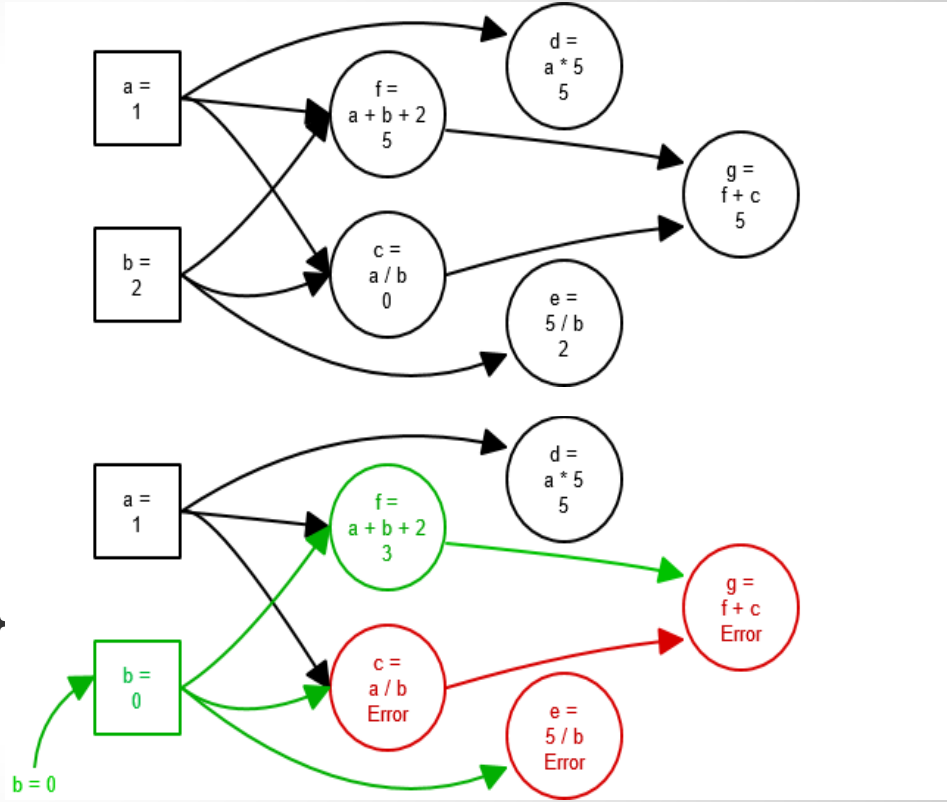


Exceptions

```
val a = Var(1L)
val b = Var(2L)

val c = Rx{ a() / b() }
val d = Rx{ a() * 5 }
val e = Rx{ 5 / b() }
val f = Rx{ a() + b() + 2 }
val g = Rx{ f() + c() }

b() = 0 // uh oh
```



Console Demo

Scala.js Demo

Exceptions Demo

Scala.js Demo 2

How

```
val a = Rx{b() + c()}
```

- `Rx.apply` pushes itself onto a thread-local stack before evaluating contents

- `b.apply`, `c.apply` look at who's on top of the stack and add the dependency

Propagation Strategy

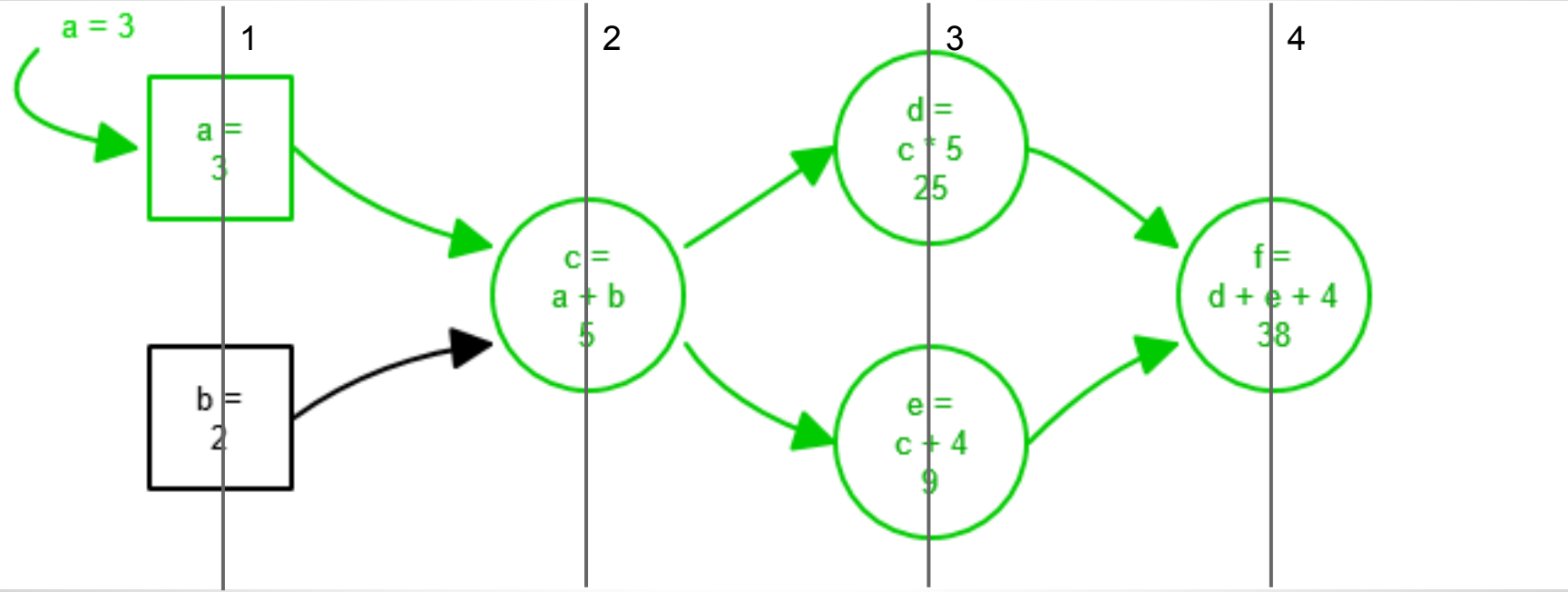
- Controlled by a `Propagator`

- When call `Var.update`, how/when do its dependencies update?

Propagation Strategy

- `Propagator.Immediate`: happens on current thread, finishes before `.update` returns
- `Propagator.ExecContext`: happens on whatever `ExecutionContext` is given, `.update` returns a `Future[Unit]`
- Both happen in roughly-breadth-first, topological order.

Topological Order



Overall Characteristics

- Dependency graph constructed at runtime
  - No need to live in a monad
  - No need to specify what the dependencies are
- No globals, only one thread-local stack
  - Easy to use as one part of a larger program.
  - Small fragments of change-propagation in a larger non-Scala.Rx world
  - Easily interops with non-Scala.Rx world

Limitations

- Dependency graph can change shape
  - `Rx`s may evaluate out of order
  - `Rx`s may evaluate more than once
- Thread local stack doesn’t play nicely with `Future`s
- `Rx` initialization is blocking
  - Can’t initialize more than one in parallel

Limitations

```
val a = Var(1) // depth 0
val b = Rx{ a() + 1 } // depth 1


val c = Rx{ // depth 1 or 2???
  if (random() > 0.5)  b() + 1
  else a() + 1
}
```

Limitations

```
val a = Rx{ ... }
val b = Rx{ Future(a()) }
```

Limitations

```
import concurrent.ExecutionContext.global
implicit val prop = {
  new Propagator.ExecContext()(global)
}
val a = Var(1)
val b = Rx{ expensiveCompute(a() + 1) }
val c = Rx{ expensiveCompute(a() + 2) }
```

Scope

- Useless in stateless web services
- Useless in pure-functional code
- Doesn’t support a rich event-stream API
- Doesn’t support channels, coroutines, async

Works on Android too!

```
// create a reactive variable
val caption = rx.Var("Olá")
// set text to “Olá”
textView <~ caption.map(text)
// text automatically updates to “Adeus”
caption.update("Adeus")
```

- Example taken from http://macroid.github.io/guide/Advanced.html
- Warning: I haven't tried it myself

What

- `Var`: reactive variables that are set manually
- `Rx`: reactive values that depend on other reactive values
- `Obs`: observes changes to reactive values and does things

Past Work

- Lots of existing FRP libraries
- Most are written in Haskell
  - Or some custom dialect of Haskell
  - Or some custom dialect of Java
- None of them interop easily with “normal” code

Future Work

- Clean up implementation
  - Internals are a big mess
  - Lots of code related to multithreading useless on ScalaJS and should be separated out
- Experiment with a persistent file backend?
  - Currently very similar to SBT's dataflow graph
  - ...but much easier to use
  - Maybe it's generic enough to be useful?

If you liked the Demo

- Scala.js - 0.5.0, by sjrd and gzm0
- Scalatags - 0.3.0
- Scala.Rx - 0.2.5
- Workbench - 0.1.2
- Workbench-Example-App

Questions?

Ask me about

- Scala.React
- Multithreaded Execution Model
- Memory Modal
- Delimited Continuations
- Running on ScalaJS