### Introduction to Rholang

Who are we?

Ahsan Fazal Full stack developer ahsanfazal.com Thomas Schoffelen Full stack developer thomasschoffelen.com

### Basics of Rholang Syntax

### Conventional programming languages

- define a function
  call that function
  function gets executed

### Conventional programming languages

define a function
call that function
function gets executed

```
1
2 function conventional (input) {
3 const output = 'Hello ' + input + '!'
4 return output
5 }
6
7 conventional('RChain')
8
```

### Conventional programming languages

define a function
call that function
function gets executed

```
1
2 function conventional (input) {
3 const output = 'Hello ' + input + '!'
4 return output
5 }
6
7 conventional('RChain')
8
```

OUTPUT

Hello RChain!

### Concurrency

### eat sandwich | watch tv

### Channels (sending and receiving)

channel!()

### for(\_ <- channel)

### for( <= channel)

# channel!() | for(\_ <- channel)</pre>

# for(\_ <- channel) | channel!()</pre>

### Matching

# new channel in { for(@x, @y <- channel) {</pre>

 $\bullet \quad \bullet \quad \bullet$ 

# } | channel!("hello", "world")

# new channel in {

## } channel!("hello")

 $\bullet \quad \bullet \quad \bullet$ 

## for(@x, @y <- channel) {</pre>

# new channel in {

 $\bullet \quad \bullet \quad \bullet$ 

### } channel!("hey", "world")

# for(@"hello", @y <- channel) {</pre>

# new channel in {

 $\bullet \quad \bullet \quad \bullet$ 

## } channel!("hello", "world")

# for(@"hello", @y <- channel) {</pre>

### Named processes

\*stdOut

### contract stdOut {}

### channel!(\*stdOut)

6"std0ut"!()



- asynchronous ()
- send (!) and receive (for loop) on channels
- convert between names and processes
  - @ = process from name
  - \* = name from process

### Recap

### Building Smart Contracts Using Rholang

•••		
×	Core ~	Core / Pages
Q +	← Pages	The Flight
I	, Meeting notes	Last modified
	, (archive) Mercury Mercury Documentation	This page attemp not finalized. Plea
	<ul><li>Namespaces</li><li>Nodes</li></ul>	At a very high l
	<ul> <li>Product requirements</li> <li>Resource Types</li> </ul>	<ul> <li>Launch of t</li> <li>Launch of n</li> <li>Launch of t</li> </ul>
	Roadmap - Draft	VM Milestones
	The Flight to Mercury	Name
	Tuplespace Notes	Roscala.Void release plan
≡	Upgrades/Updates	Roscala.Transit release plan
?	RHOC/Rev swap specification API Forward / Backward compatibility	Roscala.Primiti release plan
→	JIRA reports	Roscala.FFI- Di

### nt to Mercury

### rlikar

ed May 07, 2018 by Kelly Foster

ots to lay out the large milestones in the project. Dates in the graphical roadmap are ease refer to Milestone pages for actual dates.

C

### level, there are three key milestones in this project:

the RChain testnet - July 2018 name registry - September 2018

the RChain main net - December 2018

### **Node Milestones**

	Status	Date	Name	Status	Date
d I	RELEASED	<b>16 C</b>	Node.Hello release	RELEASED Release	22 Dec 2017
nsition I	IN DEVELOPMENT	TBD	plan node - 0.1	announcement	🗐 15 Mar 2018
mitive I	PLANNING		release plan	Release announcement	
- Draft	PLANNING		node - 0.2	RELEASED	29 Mar 2018

+

### A Rholang tutorial.

Rholang is a new programming language designed for use in distributed systems. Like all newborn things, it is growing and changing rapidly; this document describes the syntax that will be used in the RNode-0.3 release.

Rholang is "process-oriented": all computation is done by means of message passing. Messages are passed on "channels", which are rather like message queues but behave like sets rather than queues. Rholang is completely asynchronous, in the sense that while you can read a message from a channel and then do something with it, you can't send a message and then do something once it has been received---at least, not without explicitly waiting for an acknowledgment message from the receiver. Note that throughout this document the words "name" and "channel" are used interchangeably. This is because in the rho-calculus (on which Rholang is based) the term name is used, however because you can send and receive information on names, semantically they are like channels.

### Getting started

There is not an IDE for Rholang. Get started with Rholang by selecting one of the options below.

- EVAL modes. Get started with the latest version of RNode.
- Run Rholang on a web interface This web interface was created by a RChain community member.
- Write Rholang using an IntelliJ plugin This Rholang IntelliJ plugin was created by a RChain community member.

• Run Rholang on RNode - Write Rholang contracts in an editor of your choice and run them on RNode using either the REPL or

C



### rchain.cloud

YOUR CODE

1	new helloWorld in {
2	contract helloWorld(@name) = {
3	new ack in {
4	@"stdoutAck"!("Hello!", *ack)
5	for (_ <- ack) {
6	@"stdout"!(name)
7	}
8	}
9	}
10	helloWorld!("Joe")
11	}

Run

rchain.cloud C +online rholang interpreter 🎧 OUTPUT



### rchain.cloud

### YOUR CODE

1	new helloWorld in {
2	contract helloWorld(@name) = {
3	new ack in {
4	@"stdoutAck"!("Hello!", *ack)
5	for (_ <- ack) {
6	@"stdout"!(name)
7	}
8	}
9	}
10	helloWorld!("Joe")
11	}

Run

```
C
rchain.cloud
                                                                   +
                                           online rholang interpreter 🎧
     UPLOADING
     EVALUATING
     new x0 in { x0!("Joe") | for( @{x1} <= x0 ) { new x2 in {
     @{"stdoutAck"}!("Hello!", *x2) | for( _ <- x2 ) {
     @{"stdout"}!(x1) } } }
     OUTPUT
     @{"Hello!"}
     @{"Joe"}
    STORAGE CONTENTS
     COMPLETED
```



### rchain.cloud

### YOUR CODE

1	new helloWorld
2	contract hell
3	new ack in
4	@"stdoutA
5	for (_ <-
6	@"stdou
7	}
8	}
9	}
10	helloWorld!("
11	}

```
in {
loWorld(@name) = {
ł
\ck"!("Hello!", *ack)
 ack) {
it"!(name)
```

'Joe")



COMPI

### new helloWorld in { 1 2 new ack in { 3 4 for (\_ <- ack) { 5 6 7 8 9 helloWorld!("Joe") 10 11 }

```
contract helloWorld(@name) = {
   @"stdoutAck"!("Hello!", *ack)
     @"stdout"!(name)
```

```
new helloWorld in {
 1
      contract helloWorld(@name) = {
 2
        new ack in {
 3
          @"stdoutAck"!("Hello!", *ack)
 4
          for (_ <- ack) {
 5
            @"stdout"!(name)
 6
           }
 7
8
9
       }
      helloWorld!("Joe")
10
11
    }
```

OUTPUT

Hello! Joe

```
new stdoutAck2, helloWorld in {
 1
       contract stdoutAck2(@message, channel) = {
 2
          @"stdout"!(message)
 3
          channel!(0)
 4
       }
 5
      contract helloWorld(@name) = {
 6
        new ack in {
 7
          stdoutAck2!("Hello!", *ack) |
8
          for (_ <- ack) {
 9
            @"stdout"!(name)
10
11
           }
12
13
      helloWorld!("Joe")
14
15
     }
```

```
new stdoutAck2, helloWorld in {
 1
        contract stdoutAck2(@message, channel) = {
 2
          @"stdout"!(message) |
 3
 4
          channel!(0)
 5
       }
 6
       contract helloWorld(@name) = {
        new ack in {
 7
           stdoutAck2!("Hello!", *ack) |
 8
          for (_ <- ack) {
 9
            @"stdout"!(name)
10
11
           }
12
13
      helloWorld!("Joe")
14
15
    }
```

OUTPUT

Joe Hello!

### Composability

1	new calculator in {
2	new channel, valueSte
3	valueStore!(0)
4	contract calculate
5	ret!(*channel)
6	for(@"sum", @a
7	for(_ <- va
8	valueSto
9	}
10	}
11	}
12	}
13	new ret in {
14	calculator!(*ret)
15	<pre>for(object &lt;- ret)</pre>
16	object!("sum",
17	for(@result
18	@"stdout
19	}
20	}
21	}
22	}
23	}

```
core in {
cor(ret) = {
    |
arg1, @arg2, ack <= channel) {
alueStore) {
    pre!(arg1 + arg2) | ack!(arg1 + arg2)</pre>
```

```
|
] {
] 1, 3, *ret) |
: <- ret) {
:"!(result)
```

1	new calculator in {
2	new channel, valueSt
3	valueStore!(0)
4	contract calculate
5	ret!(*channel)
6	for(@"sum", @a
7	for(_ <- va
8	valueSto
9	}
10	}
11	}
12	}
13	new ret in {
14	calculator!(*ret)
15	for(object <- ret
16	object!("sum",
17	for(@result
18	@"stdout
19	}
20	}
21	}
22	}
23	}

```
tore in {
tor(ret) = {
  |
  arg1, @arg2, ack <= channel) {
  alueStore) {
  pre!(arg1 + arg2) | ack!(arg1 + arg2)</pre>
```



```
for(@"sum", @arg1, @arg2, ack <= channel) {</pre>
   for(_ <- valueStore) {</pre>
      valueStore!(arg1 + arg2) | ack!(arg1 + arg2)
```

```
@"stdout"!(result)
```



```
for(@"sum", @arg1, @arg2, ack <= channel) {</pre>
   for(_ <- valueStore) {</pre>
      valueStore!(arg1 + arg2) | ack!(arg1 + arg2)
```

```
@"stdout"!(result)
```

### **Built-in Security**

### Principle of Least Authority

### Revocable Forwarder

Source Object





1	new MakeRevokableFor
2	contract MakeRevol
3	new port, kill,
4	ret!(*port, *
5	forwardFlag!(1
6	contract port
7	for (@status
8	forwardFla
9	match stat
10	}
11	}
12	for (_ <- kil]
13	forwardFlag
14	}
15	}
16	}
17	}

```
;!(false)
```

### new GetAccount, target in { 1 contract GetAccount(callback) { 2 new ret in { 3 4 5 6 } 7 8 9 } 10 }

```
MakeRevokableForwarder!(*target, *ret) |
for(@port, @kill <- ret) {</pre>
   callback!(port)
```

 $\equiv$ 

### Dining philosophers and deadlock

=

```
1 new philosopher1, philosopher2, north, south
     north!(*knife) |
 2
     south!(*spoon)
 3
     for (@knf <- north) { for (@spn <- south)</pre>
       philosopher1!("Complete!") |
 5
       north!(knf) |
 6
       south!(spn)
     } } |
 8
     for (@spn <- south) { for (@knf <- north)</pre>
 9
       philosopher2!("Complete!") |
10
       north!(knf) |
11
       south!(spn)
12
13
     14 }
```

The dining philosophers problem has two philosophers that share only one set of silverware. Philosopher1 sits on the east side of the table while Philosopher2 sits on the west. Each needs both a knife and a spoon in order to eat. Each one refuses to relinquish a utensil until he has used both to take a bite. If both philosophers reach first for the utensil at their right, both will starve: Philosopher1 gets the knife, Philosopher2 gets the spoon, and neither ever lets go.

Here's how to solve the problem:



**\$** suggest a change

+

, knife, spoon in {	
{	
{	
	Run example

C

### developer.rchain.coop