Contracts, Composition, and Scaling

The Rholang specification

Draft version 0.2

27 Feb 2018

Lucius Gregory Meredith
Jack Pettersson
Gary Stephenson
Michael Stay
Kent Shikama
Joseph Denman

Intended audience: Developers using the RChain Platform; developers creating or maintaining the RChain Platform; language designers; concurrency theorists; blockchain and cryptocurrency enthusiasts

As playful as the natural world is, natural actions have natural consequences.

The contract between client and service provider

can mean the difference between flourishing and failing.

Yet, with stakes this high nature scales naturally. Gracefully. Beautifully.

Instinctively, we know how nature does this. Every child can see the tree in the leaf. Yet, we forget, and we only seem to remember, when we remember at all, at the crucial moment when we need to build systems at scale, and the systems we have built are failing. Like now.

Introduction and motivation

The need for social contracts

Bitcoin and its proof-of-work based blockchain pointed the way towards a trustless medium for financial and information-based exchange by which this generation can address the egregious failures of the financial systems and the states that allegedly regulate and govern them. By eliminating the role of the trusted 3rd party, this technology and the generation that adopts it has no use for financial institutions that are too big to fail, nor do they have to kowtow to government agencies that subsidize systemic irresponsibility with taxpayer funds. They can realize effective alternatives without having to stomach the absurdity that there's no other way, nor wait for public institutions to come up to speed on the real potential of Internet technologies. However, the incumbent financial services layer is rich with complex financial instruments, a fact not lost on Ethereum and others working on emerging smart contract technologies.

These new offerings enrich the basics of the blockchain tool set with computationally complete platforms in which to re-engineer, and more importantly reimagine, the layer of financial instruments. Meanwhile, efforts like Factom recognize that blockchain isn't just for keeping ledgers of who has how many quatloos. Records of land trust are frequently massively abused in many countries, and Factom is putting this information on the blockchain so that all can see a public record and attestation of ownership.[1]

Likewise, RChain recognizes that information flow goes hand in hand with financial instrumentation. Further, it's not just about provenance. Contractual obligations in the blockchain enabled world are just as much about the when, where, and how of disclosure of sensitive information. RChain introduces these ideas in a relatively low-risk setting of social networks, but this is primarily because we believe that establishing trust in the technology in low-risk settings, at Internet-wide scale is the right path to adoption in markets where there is more risk, such as freelance work, or dating, the exchange of medical records, or self-determination via the Internet.

Further, RChain recognizes that a computationally complete platform needs two key elements to provide realistic and scalable services. First and foremost it needs a clear computational semantics grounded in formal methods. ML and Haskell are two prominent and popular languages enjoying this kind of grounding and their success is in no small measure derived from their foundations. Of course, these languages were conceived when computing was largely a desktop phenomenon, not the fabric of global society, taking place

[1] More accurately, Factom is putting hashes of land trust records into their blockchain and anchoring this to the bitcoin blockchain.

on billions of interconnected devices, with a communication topology that is continually dynamically reconfiguring. As such, neither language, nor the computational models on which they are based are particularly well suited for this new purpose. No, the computational semantics demanded must be ones that suit the requirements of this time and place — which is why RChain's contract language is based on the mobile process calculi, which provide, by design, a mathematical model of today's computing infrastructure.

Secondly, but of equal weight, the computational platform must provide a means to constrain, probe, and examine contracts, which became especially apparent in the wake of the exploit of "The DAO". The accepted way to do this in modern programming language design and programming language semantics is with types. RChain's contract language comes equipped with a modern, behavioral typing system. This system allows parties seeking to engage contractually with a means to probe the contractual obligations and guarantees in an automated way. This gives potential participants a chance to determine if becoming party to a particular contract is really the right choice.

Additionally, when a new technology makes it easy to forge or manufacture a useful artifact, what follows shortly after is a proliferation of such artifacts. Making it easy to create and execute smart contracts will invariably result in an abundance of smart contracts. Think about this for a minute. Smart contract is just another name for code. If we've learned nothing else from source control systems from CVS to SVN to github, we should have learned that code is the dark matter of the Internet. Whether inside the corporate firewall or outside in the land of open source, searching for code that does just what needs to be done is a dark art, best left to wizards. Already, technologies like Hoogle are aimed at searching on the basis of type information. RChain's behavioral types makes it possible to take that idea much, much further — to search code on the basis of how it is shaped and what it does — not just what strings, keywords or phrases might happen to show up somewhere in its text.

Both of these features address scaling in several dimensions. Choosing a semantics that naturally reflects how computing actually happens in the Internet is of paramount importance. Ethereum's choice of VM that imposes global serialization across Internet scale contracts is precisely why Ethereum V 1.0 won't scale. Well, that and the fact that proof-of-work doesn't scale, either. At DevCon1, Vitalik's talk on scaling Ethereum admits just this. Both his proposals for changing the architecture and for the corresponding changes to the contract language that codes to it are several steps in the direction of the model RChain has had from the beginning.

Where RChain's choices really begin to shine, however, are in the features that emerge from having a behavioral type system. One of the key aspects of scaling is not just how fast or how many, but who and why. Which agents will use the system and for what purpose? It won't be long before the major corporate agencies realize that blockchain — as currently rendered in running code — is simply a much, much slower way to do things than their internal systems. It has virtually no use behind the corporate firewall. It's real value is in the public domain

between private agents. Add to this that the fact that it's brand new technology with all manner of warts and risks and interest will wane very soon. Solving the speed and throughput problems alone won't address this latter concern.

Behavioral typing of social contracts means that — for the first time — we have a technology that can enforce information flow and liveness properties at Internet scale. This is the natural next step in trustlessness. Agents who never had a relationship can become party to an automated contract with certainty that that contract will not leak sensitive information, and that that contract will — under appropriate guarantees — reliably execute mission-critical transactions. These kinds of guarantees will bring corporate and governmental agency to the blockchain — because these agencies are built on providing mission-critical services. But bringing these players to the table effectively shifts the balance of power back to the public that maintains such a valuable infrastructure. That's scaling of a completely different color, to mix metaphors.

Examples of social contracts

Let's bring these ideas back to earth, and begin somewhere small by making a contract that holds a resource, like a balance, and allows clients to get and set the value of the resource. Such Cell-like behavior is the "hello world" of the blockchain.

```
contract Cell( get, set, state ) = {
  for( rtn <- get; @v <- state ) {
    rtn!( v ) | state!( v ) | Cell( *get, *set, *state )
  } |
  for( @newValue <- set; _ <- state ) {
    state( newValue ) | Cell( *get, *set, *state )
  }
}
```

This takes a channel for get requests, a channel for set requests, and a state channel where we will hold the resource. In parallel it waits on the get and set channels for client requests. It joins an incoming client with a request against the state channel. This join does two things. Firstly, it removes the internal state from access while this, in turn, serializes get and set actions, so that they are always operating against a single consistent copy of the resource. Of course, the serialization is still subject to the non-deterministic arrival order of client requests. But, there will be some serialization of those requests providing a single history of accesses and updates against the state.

A get request comes with a rtn channel where the value in state will be sent. Since the value has been taken from the state channel, it is put back, and the Cell behavior is recursively

invoked. A set request comes with a new value, which is published to the state channel (the old value having been stolen by the join). Meanwhile, the Cell behavior is recursively invoked.

We can instantiate and run the Cell with a private state channel, called current, an initial value, initial by

```
new current in { Cell( *get, *set, *current ) | current!( initial ) }
```

and we can wrap that up in a Wallet contract that is parametric in the get and set channels and the initial value.

```
contract Wallet( get, set, @initial ) = {
  new current in { Cell( *get, *set, *current ) | current!( initial ) }
}
```

Finally, we can instantiate a Wallet.

```
Wallet( *get, *set, 1000.00 )
```

That's a simple Wallet contract that can hold a balance (or other kind of resource), and allow the balance to be updated.

One less desirable aspect of this implementation is that it will accumulate threads. To see this, consider what happens when servicing a client's request on the get channel. The Cell is recursively invoked; however, there is still a thread waiting to service a set request, and another such thread will be launched on the recursive call to Cell. A large imbalance of get (resp. set) requests and this implementation will run out of memory. A safer implementation would use the select construct

```
contract Cell( get, set, state ) = {
  select {
    case rtn <- get; @v <- state => {
      rtn!( v ) | state!( v ) | Cell( *get, *set, *state )
    }
    case @newValue <- set; _ <- state => {
      state!( newValue ) | Cell( *get, *set, *state )
    }
  }
}
```

This implementation may be substituted into the Wallet contract without any perturbation to that code context. However, when it is run only one of the threads in Cell can respond to the client request. It's a race, and the losing thread, be it getter or setter, is killed. This way,

when the recursive invocation of Cell is called, the losing thread is not hanging around, yet the new Cell process is still able to respond to either type of client request.

For programmers who prefer a more object-oriented style with rich message structure, there is yet a third option that uses only one client request channel, and dispatches on the type of the message received on the channel.

```
contract Cell( client, state ) = {
 for( request <- client; @v <- state ) {
  request match {
   case @("get", rtn) => {
    rtn!( v ) | state!( v ) | Cell( *client, *state )
   }
   case @("set", @newvalue) => {
    state!( newValue ) | Cell( *client, *state )
   }
  }
 }
}
```

This implementation would require a change to the Wallet contract. Either the Wallet contract has to turn requests on the get and set channels into messages

```
contract Adapter( get, set, client ) = {
 select {
  case rtn <- get => {
   client!( ("get", *rtn) ) |
   Adapter( *get, *set, *client )
  }
  case @newValue <- set => {
   client!( ("set", newValue) ) |
   Adapter( *get, *set, *client )
  }
 }
}

contract Wallet( get, set, @initial ) = {
 new client, current in {
  Adapter( *get, *set, *client ) |
  current!( initial ) |
  Cell( *client, *current )
 }
}
```

or it has to pass along to clients the change in the contractual interface.

```
contract Wallet( client, @initial ) = {
  new current in { Cell( *client, *current ) | current!( initial ) }
}
```

Even with this basic example we can see many of the salient features of the language. Concurrent execution, asynchronous message-passing, and pattern matching are woven together into a simple, easy-to-understand language.

How social contracts differ from Ethereum's smart contracts

To begin with, Ethereum's contracts are internally sequential. In fact, the entire call-chain stemming from a point of entry at a single contract will have a global serial order. Think about this in terms of supply chain management. Does Ford Motor Company want to serialize the tire supply with the chassis or engine or electrical supply? Does Boeing want to serialize the fuel system supply with the lighting or interior seating supply? Businesses are made and broken on efficiencies stemming from being able to manage processes in parallel, and coordinate them concurrently. Yet, surely Ford and Boeing could greatly benefit from a smart contract based supply chain management system. Just as with Haskell or ML, the model chosen doesn't fit the domain.

In point of fact, an earlier technology, business process modeling, already explored just this application. Microsoft's Biztalk, as well as standards like BPEL, BPML, W3C Choreography, to name a few all concluded that concurrency was the currency, so to speak, and opted to choose the mobile process calculi as their semantic foundation. The paradigmatic application example in business process modeling is supply chain management.

Language specification

RChain's social contract language

RChain offers a typed contract language that provides a semantics naturally suited for decentralized, distributed computing. It is built around communicating mobile processes.

Syntax

The specifics of the Rholang syntax are in flux; expect them to change before the Mercury release. In particular, the syntax used on the left-hand side of an arrow is likely to change; we may include a linear version of "contract"; we may include syntax for denoting intent to pay for compute versus intent to pay only for storage.

Rholang is a friendlier version of a smaller language, the rho calculus, which in turn is a reflective higher-order variant of the asynchronous polyadic pi calculus. We will begin with the syntax and semantics of rho calculus and then give the syntax and semantics of Rholang in terms of the smaller calculus.

Rho calculus syntax

```
M,N ::= 0          // nil or stopped process

    | for( x1 <- y1; ... ; xN <- yN ) P        // input guarded agent

    | x!( P )    // output

    | M + N    // summation or choice

P,Q  ::= M         // "normal" process

    | *x         // dereferenced or unquoted name

    |  P | Q     // parallel composition

x, y ::= @P          // name or quoted process
```

Free and bound names

FN(0) = {}

FN(*x) = { x }

FN(for(x1 <- y1; ... ; xN <- yN)P) = { y1, ... , yN } U FN(P) \ { x1, ... , xN }

FN(x!(P)) = { x } U FN(P)

FN(M+N) = FN(M) U FN(N)

FN(P|Q) = FN(P) U FN(Q)

Structural equivalence

Structural equivalence is the smallest congruence, $=_S$, such that

- (P,|,0) form a commutative monoid
- (P,+,0) form a commutative monoid
- If $=_N$ denotes name-equivalence, then $=_S$ includes the alpha-equivalence using $=_N$

Name equivalence

Name equivalence is the smallest equivalence on names such that

$P =_S Q \Rightarrow @P =_N @Q$

Semantic versus syntactic substitution

The substitution used in α-equivalence is really only a device to formally recognize that binding occurrences do not depend on the specific names. It is not the engine of computation. The proposal here is that while synchronization is the driver of that engine, the real engine of computation is a semantic notion of substitution that recognizes that a dereferenced name is a request to run the process whose code has been bound to the name being dereferenced. Formally, this amounts to a notion of substitution that differs from syntactic substitution in its application to a dereferenced name.

$$*x \{ @Q / @P \} = Q \text{ if } x =_N @P \text{ and } *x \text{ otherwise}$$

Reduction relation

In what follows, => is implication and -> is "reduces to":

comm: xi $=_N$ xi' => R + for(x1 <- y1; ... ; xN <- yN)P + S | x1'!(Q1) | ... | xN'!(QN) -> P{ @Q1/y1, ... , @QN/yN }
par: P -> P' => P|Q -> P'|Q
struct: P = P', P' -> Q', Q' = Q => P -> Q

Rholang syntax

We have added comments to the syntax to indicate something of the intended semantics; future versions of this document will include explicit desugarings of these productions into the rho calculus.

```
<process> ::= "Nil"
        // Grouping
        | "{" <process> "}"

        // Equivalent to "for" "(" [ <names> ] "<=" <name> ")" <process>
        | "contract" <name> "(" [ <names> ] ")" = <process>

        // Receiving; only boolean processes allowed in if clause.
        | "for" "(" <receipt> ")" <process>

        // Exactly one branch proceeds.  Choice is nondeterministic.
        | "select" "{" <branch>* "}"

        // Destructuring
        // If the name does not match any of the cases, the entire process is equivalent to Nil
        | "match" <process> "{" <case>* "}"

        // Evaluates to a Boolean.  Desugars to
        // match <process> {
        //   case <process> => true
        //   case _ => false
        // }
        | <process> "matches" <process>

        // Boolean destructuring.  Desugars to
        // match <process> {
        //   case true => <process>
        //   case false => <process>
        // }
        | "if" "(" <process> ")" <process> [ "else" <process> ]

        // Send
        | <name> <send>"(" [ <processlist> ] ")"

        // Contract invocation
        | <name> "(" [ <processlist> ] ")"
```

```
        | <process> "|" <process>
        | "*" <name>

        // New name in the @private namespace.
        // Private names have no visible internal structure.
        | "new" <vlist> "in" <process>

        | <primitive>
        // Syntactically allowed with arbitrary processes, but compiler complains if
        // the processes aren't of the right type.
        | <process> <binop> <process>
        | <unop> <process>

        | <collection>
        | <procfield>

<names> ::= <typedname> [ "," <typedName> ]*
<typedname> ::= <name> [ ":" <process> ]
<name> ::= <var> | <quote>
// The underscore is for discarding the binding from the match
<var> ::= <identifier> | "_"
<quote> ::= "@" <process>

// Wait on a product of channels
<receipt> ::= <linreceipt> | <nonlinreceipt>
<linreceipt> ::= <names> <linarr> <name>
                                [ ";" <names> <linarr> <name>]* [ <ifclause> ]
<nonlinreceipt> ::= <names> <nonlinarr> <name> [ <ifclause> ]
<arrow> ::= <linarr> | <nonlinarr>
<linarr> ::= "<-"
// Linear-receive-then-send, sequential receive, replicated receive
<nonlinarr> ::= "<!" | "<<" | "<="
// The syntax allows any process, but the compiler complains if it is not a Boolean
<ifclause> ::= "if" <process>

<send> ::= "!" | "!!"

<branch> ::= <linreceipt> "=>" <process>

<case> ::= "case" <process> "=>" <process>

<processlist> ::= <process> [ "," <process> ]*
```

```
<vlist> ::= <vdecl> [ "," <vdecl>]*
<vdecl> ::= <simplevdecl> | <iopairvdecl>
<simplevdecl> := <var> [ ":" <type>]
<iopaorvdecl> := "(" <var>, <var> ")" ":" "iopair" [<type>]

<primitive> ::= <boolean>
        // Signed 64-bit integer
        | <int64>
        // Delimited by apostrophes or quotation marks
        | <string>
        | "DateTime" "(" <iso8601> ")"
        // Delimited with backticks
        | <uri>

<binop> ::=
        // Structurally equivalent processes
        "=="
        | "!="
        // Boolean operators
        | "and" | "or"
        // Integers, strings, and datetimes
        |  "+"
        // Integers and strings
        | "*" | "%"
        // Integers
        | "-" | "/"
        // All ordered types
        |  "<="  |  "<"  |  ">="  |  ">"
        // Bitfields
        |  "bitand"  |  "bitor"

<unop> ::=
        // Boolean
        "not"
        // Bitfield
        | "bitnot"
        // Integer
        | "+" | "-"

<collection> ::= <list> | <tuple> | <set> | <map>
<list> ::= "[" [ <processlist> ] "]"
<tuple> ::= "(" [ <processlist> ] ")"
```

<set> ::= "Set" "(" [<processlist>] ")"
<map> ::= "{" [<kvlist>] "}"
<kvlist> ::= <kvpair> ["," <kvpair>]*
<kvpair> ::= <process> ":" <process>

<procmeth> ::= <process> "." <identifier> "(" [<processlist>] ")"

Spatial types

The grammar for spatial types is the grammar for processes extended by the following productions:

<process> ::= "private"
| "~" <process>
| <process> "&&" <process>
| <process> "||" <process>
| "=" <process>

Rholang syntactic sugar

Replication

Notice that when the reduction rules of the semantics section are used, the process

x!(for(y <- x){ x!(*y) | *y } | P) | for(y <- x){ x!(*y) | *y }

reduces to

P | P | ...

In other words, this expression constitutes an implementation of replication, and thus the code context for P represents a concurrent version of the famous Y-combinator for the lambda-calculus.

It is well known that recursion and replication are inter-definable, and providing a version of recursion from replication is a useful and instructive exercise. In this connection, notice that it is always possible to convert an input guarded process expression into one that will receive its continuation from an outside source. That is,

for(y <- x) { P } = { k(P) | for(y <- x; p <- k) { *p } }

This observation gives rise to a conversion of processes to a normal form we call continuation-saturated. Continuation-saturation has many uses, both in terms of storing

processes for long term execution, but also in terms of supporting reversible computation. Continuation-saturation is the key to a compilation strategy that turns every Rholang process into a reversible computation. This affords natural semantics for transactions with rollback. For purposes of this discussion, however, note that by factoring processes into their continuation-saturated normal form it is possible to define a version of replication that only works for input-guarded processes. For the ambitious student of the language, it is useful and enlightening to work out this version of replication.

Persistent I/O

When sending data over channels, the default behavior is that both the data and the consumer's continuation are ephemeral, i.e. that the data disappears from the channel after being read, and the consumer of the data reads from the channel exactly once. This is the behavior that we've seen so far, and exactly the same as in the underlying calculus. However, we also provide syntactic sugar for when the data, continuation, or both, are persistent, which gives rise to quite interesting and practical patterns that allow channels to emulate e.g. data streams and memory locations.

This section describes the different combinations and explains how they are desugared to the default behavior. In the interest of clarity, we consider all pairs of the form:

data consumer | *data producer*

Data ephemeral, continuation ephemeral

This is the standard process calculus expression where both the continuation and the data are ephemeral. That is, the channel is used exactly once by the consumer, and the data is removed from the channel once it's read:

```
for(v <- channel) { P } | channel!(Q)
```

Data persistent, continuation ephemeral

This means that the channel is used as an append-only set of values, i.e. each value will remain in the channel even after it has been read.

```
for(v <- channel) { P } | channel!!(Q)
```

However, sometimes it is useful for the consumer to only peek at a value in a channel, even if the producer didn't specify that it should persist. This is achieved by using the receive-then-resend operator `<!`, as here:

```
for(v <! channel) { P } | channel(Q)
```

It is syntactic sugar for

```
for(v <- channel) { channel!(*v) | P } | channel(Q)
```

Of course, these two can be combined. Both the producer and the consumer could want to make sure that the value will stay in the channel, which they express by:

```
for(v <! channel) { P } | channel!!(Q)
```

Desugaring this using the rules given in the previous two examples, we see that this is sugar for:

```
for(v <- channel) { channel!(*v) | P } | channel!!(Q)
```

Data ephemeral, continuation persistent

If the consumer wants to listen to a channel as a stream, it wants to execute the same continuation for every message that is received. In other words, the continuation should persist. The arrow <= indicates that the for production should be replicated.

```
for(v <= channel) { P } | channel!(Q)
```

Data persistent, continuation persistent

Finally, we come to the case where both the data and the continuation should persist. The following idiom expresses an unbounded computation:

```
for(v <= channel) { P } | channel!!(Q)
```

That is, the same value should be sent over the channel infinitely many times, and it should be read and passed to the continuation infinitely many times.

This may be what a programmer intends, since it is not a liveness problem. Note that

```
for(v <= channel) { P } | channel!!(Q) | channel!!(Q')
```

is a process in which both the data values Q and Q' as well as the continuation P persist.

On the other hand, a programmer may want a process where the continuation waits for a signal on some other channel, and only then handles the data:

```
for (_ <= signal) { for (v <- channel) { P } | channel!(Q) }
```

Every time a message is received on the signal channel, a new ephemeral continuation and a new ephemeral data value is produced.

Serial communication

If a finite list is known in advance, it can be sent in a single message:

```
for (@list <- channel) { print!(list) } | channel!([4, 5, 6])
```

If the list is generated one term at a time, the whole process becomes much more complicated:

```
new printEach, iterate, channel, ack in {
        contract printEach() = {
                for (@item, @done <- channel) {
                        print!(item, *ack) |
                        if (done) { printEach() } else { Nil }
                }
        } |

        contract iterate(@list, @i, @limit) = {
                channel!(list[i], i < limit) |
                for(_ <- ack) {
                        if (i < limit) { iterate(list, i + 1, limit) } else { Nil }
                }
        } |

        printEach() | iterate([4, 5, 6], 0, 3)
}
```

This pattern is common enough that it merits some sugar. The process

```
for ( v, ack << channel ) { P }
```

desugars to

```
for (ack <- channel) {
        ack!() |
        new left in {
                for (_ <= left) = {
                        for (v, keepGoing <- channel) {
                                P |
                                if (keepGoing) { left!() } else { Nil }
                        }
                }
        }
```

where left and done are fresh with respect to P. The desugaring first waits for a channel "ack" to acknowledge receipt of messages on, then waits for pairs consisting of a value v and a boolean done. P then handles v and acknowledges receipt, while the rest of the process either waits for another pair or quits, depending on the "done" flag.

Similarly, Lists have an iterate method that interacts with the code above. The process

```
Coll.iterate
```

is defined to be

```
contract iterate(channel) = {
        new ack, right in {
                channel!(*ack) |
                for (_ <- ack) {
                        for(@i, @limit <= right) {
                                channel!(Coll[i], i < limit) |
                                for (_ <- ack) {
                                        if (i < limit) { right!(i + 1, limit) } else { Nil }
                                }
                        } |
                        right(0, Coll.size)
                }
        }
}
```

so that the process

```
for ( v, ack << channel ) { print!(*v, *ack) } | [4, 5, 6].iterate(*channel)
```

invokes "print" three times, once each on 4, 5, and 6. Sets and Maps have methods that return lists of data on which the iterate method can be called.

Primitives and foreign functions

All values, such as booleans, integers, strings, datetimes, URIs, and collections of these, are processes. This means that @57, @"joe", @(123, false), @{"hello" => "world"}, etc., are all public names on which processes can communicate. Note that the syntax rules prevent a process P from listening on @P, so there is no sense in which @57 communicates with the number 57.

Functions can be embedded as processes that take input and a return channel, then send the result on the return channel. For example, the JavaScript function

```
function inc(x) { return x+1; }
```

could be embedded in Rholang as

```
contract inc(@x, ret) = { ret!(x + 1) }.
```

There is no explicit provision for foreign functions in the Rholang syntax. Instead, platforms may expose functionality as special public names. Note that we can attenuate the authority these public names provide by using the type system to restrict the names that a piece of

code can send on. For example, if we insist that a piece of code only communicate on names in the @private namespace, then the only way the code could use a public name is by means of a proxy process listening on a private name that forwards to the public name. The proxy is then in a position to filter messages from the confined code.

Semantics

The operational semantics of Rholang can be given as a multisorted nominal Lawvere theory describing the graph of terms and rewrites. The two sorts of the theory are T for terms and R for rewrites. Each grammatical production corresponds to a function symbol taking some number of subterms to a larger term; for example, the production <process> | <process> corresponds to a function symbol

$$|:T^2 \to T$$

taking (P, Q) to P|Q. Each reduction rule corresponds to a function symbol taking some number of subterms to a rewrite from the initial context for those terms to the final context; for example, the comm rewrite rule corresponds to a function symbol

$$comm:T^4 \to R$$

taking (P, Q, R, S) to the rewrite

$$\text{for (@S <- @P) \{ Q \} | @P!(R) -> Q\{ @R / @S \}.}$$

Structural equivalence rules correspond to equations between function symbols; for example,

$$P \mid Q \equiv Q \mid P.$$

Guidance for implementations

Ignoring the nuances around the structure of names, here is a reasonable rendering of the core concurrency semantics into Scala code using the Akka actor framework. The interpretation is a function with signature

$$[\![-]\!](-): \text{Rholang} \times \text{Map(Symbol, Queue)} \to \text{Scala}.$$

Rholang	Scala
0	{ }
x!(y1, ..., yn)	[[x]](m) ! ([[y1]](m), ..., [[yn]](m))

for ((y1, ..., yn) <- x) { P }	for ((y1, ..., yn) <- [[x]](m)) {
	  [[P]](m)(y1, ..., ym)
	}

P | Q	spawn { [[P]](m) }; spawn { [[Q]](m) }

new x in P	{ val q = new Set(); [[P]](m + ("x" <- q)) }

Rate limiting

The computational resources available to a contract need to be limited to the resources the contract has fairly obtained. To that end, every computationally significant action expressible in Rholang has a corresponding cost that the contract must pay prior to executing the action.

Ideally, we'd like to express the cost model as an extension of the core calculus, so that we can reason about complexity formally. We introduce a Labeled Transition System (LTS) for the RHO calculus. We then extend the LTS with cost semantics and, finally, introduce a built-in rate-limiting mechanism that dynamically "meters" Rholang contracts.

Labeled transition semantics

Labeled Transition Systems are useful tools for examining the linear-time evolution of a system while retaining information about the system's history. The labeled transition relation for rho-calculus is of the form

$$A \vdash P \xrightarrow{\ell} Q$$

where A is a finite set of names and $fn(P) \subseteq A$. The above relation is read: "In a state where the names A are known to P, P can do ℓ to become Q. Labels, $\ell \in \mathcal{L}$ are defined by the grammar

$\ell ::= \tau$	internal transition
ρ	unquote $@Q$ to Q
$x!(@Q)$	output $@Q$ on x
$for(@Q \leftarrow x)$	input $@Q$ on x

The free names of labels are $fn(\tau) = \varnothing = fn(\rho)$, $fn(x!(@Q)) = \{x, @Q\} = fn(for(@Q \leftarrow x))$. Familiar readers will recognize the addition of the drop label, ρ, as our first departure from transition semantics as formulated for the π-calculus. Notice that the drop label has no bound names.

$$\text{LIFT}: \frac{}{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0} \qquad \text{IN}: \frac{}{A \vdash for(z \leftarrow x)P \xrightarrow{for(@Q \leftarrow x)} P\{@Q/z\}}$$

$$\text{SUM}: \frac{A \vdash P \xrightarrow{\ell} P'}{A \vdash P + Q \xrightarrow{\ell} P'} \qquad \text{TAU}: \frac{}{A \vdash \tau.P \xrightarrow{\tau} P}$$

$$\text{COMM}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P' \qquad A \vdash Q \xrightarrow{x!(@Q)} 0}{A \vdash P \xrightarrow{\ell} P'}$$

$$A \vdash P|Q \xrightarrow{\tau} P' \qquad \text{PAR} : \frac{}{A \vdash P|Q \xrightarrow{\ell} P'|Q}$$

$$\text{RHO} : \frac{}{A \vdash *q \xrightarrow{\rho} Q}$$

With labels as reference points, some spatiotemporal relations become simpler to express. For example, a partial trace can be written

$$A_1 \vdash P_1 \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P_{n+1}$$

and expressed formally,

$$\exists P_2, ..., P_n, A_2, ..., A_n. \forall i \in 1, ..., n. A_{i+1} = A_i \cup fn(\ell_i) \wedge A_i \vdash P_i \xrightarrow{\ell_i} P_{i+1}$$

In the case $fn(P) \subseteq A$, a partial trace of P with respect to A is just

$$\text{ptrn}_A(P) = \{\ell_1, ..., \ell_2 \mid \exists P'. A \vdash P \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P'\}$$

Cost semantics

Generally speaking, a cost model is a function $\mathcal{M} : \mathcal{L} \to \mathbb{Q}^+$which assigns to each label a positive rational number reflecting an abstract interpretation of the complexity of the computation represented by the label. The cost of a single transition then is expressed as $\mathcal{M}(\ell) = k_\ell \in \mathbb{Q}^+$. We derive a cost semantics from the rules presented above, where judgements are of the form

$$\text{NORM} : \frac{A \vdash P \xrightarrow{\ell} Q}{A \vdash k_P = \mathcal{M}(\ell) + k_Q}$$

and may be read: “In a state where the names A may be known to P, if P can do ℓ to become Q, then the cost of P is the cost of ℓ plus the cost of Q. The cost of P is calculated recursively by summing the cost of ℓ and the cost of the continuation Q.

$$\text{LIFT} : \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0}{A \vdash k_{x!(Q)} = \mathcal{M}(x!(@Q))} \qquad \text{IN} : \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P'}{A \vdash k_P = \mathcal{M}(for(@Q \leftarrow x)) + k_{P'}}$$

$$\text{SUM} : \frac{A \vdash P + Q \xrightarrow{\ell} P'}{A \vdash k_{P+Q} = \mathcal{M}(\ell) + k_{P'}} \qquad \text{TAU} : \frac{A \vdash \tau.P \xrightarrow{\tau} P}{A \vdash k_{\tau.P} = \mathcal{M}(\tau) + k_P}$$

$$\text{COMM} : \frac{A \vdash P|Q \xrightarrow{\tau} P'}{A \vdash k_{P|Q} = \mathcal{M}(\tau) + k_{P'}} \qquad \text{PAR} : \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q}{A \vdash k_{P|Q} = \mathcal{M}(\ell) + k_{P'|Q}}$$

$$\text{RHO} : \frac{A \vdash *q \xrightarrow{\rho} Q}{A \vdash k_{*q} = \mathcal{M}(\rho) + k_Q}$$

The cost of an A-partial trace with n transitions is simply the sum of the cost of the labels delimiting the trace.

$$\text{ptr}_A(P)_k = \sum_{i=1}^{n} \mathcal{M}(\ell_i)$$

These particular cost judgements aren't rules for statically estimating the cost of a contract, but rules for inferring the dynamic cost of a contract. With these in place, a formal notion of resource can be introduced.

Rate limiting semantics

On RChain, the resources available to a contract are denominated in "phlogiston" – a metric similar to Ethereum's "gas". During initialization, an initial quantity of phlogiston, $ph \in \mathbb{Q}^+$ is allocated to the contract. From the initial balance, each sub-process in the contract is allocated some quantity of phlogiston.

The phlogiston balance of P is denoted P_{ph} and, for every transition in P, some quantity of phlogiston representing the cost of the transition is deducted. Resource consumption is quantified using judgements of the form

$$\text{NORM} : \frac{A \vdash P \xrightarrow{\ell} Q}{A \vdash Q_{ph} = P_{ph} - \mathcal{M}(\ell)}$$

where P_{ph} is the phlogiston balance of P before the transition , and Q_{ph} is the phlogiston balance of Q after the transition. Having established cost semantics presented in the previous section, the semantic rules for rate limiting are straightforward:

$$\text{LIFT}: \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0}{A \vdash 0_{ph} = x!(Q)_{ph} - \mathcal{M}(x!(@Q))} \qquad \text{IN}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P'}{A \vdash P'_{ph} = P_{ph} - \mathcal{M}(for(@Q \leftarrow x))}$$

$$\text{SUM}: \frac{A \vdash P + Q \xrightarrow{\ell} P'}{A \vdash P'_{ph} = P_{ph} + Q_{ph} - \mathcal{M}(\ell)} \qquad \text{TAU}: \frac{A \vdash \tau.P \xrightarrow{\tau} P}{A \vdash P_{ph} = \tau.P_{ph} - \mathcal{M}(\tau)}$$

$$\text{COMM}: \frac{A \vdash P|Q \xrightarrow{\tau} P'}{A \vdash P'_{ph} = P_{ph} + Q_{ph} - \mathcal{M}(\tau)} \qquad \text{PAR}: \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q}{A \vdash P'_{ph} = P_{ph} - \mathcal{M}(\ell)}$$

$$\text{RHO}: \frac{A \vdash *q \xrightarrow{\rho} Q}{A \vdash Q_{ph} = *q_{ph} - \mathcal{M}(\rho)}$$

Halting conditions

The rules above have an implicit assumption that for any transition to occur along $A_1 \vdash P_1 \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P_{n+1}$, the transition labeled by ℓ_i is enable because the phlogiston balance of P_{i+1} is greater than or equal to zero. In implementation, this condition is held as an invariant that must be checked and satisfied prior to executing the next transition. The following rules define halting conditions based on the phlogiston balance of processes:

$$\text{ERR LIFT}: \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0 \qquad 0_{ph} < 0}{A \vdash x!(Q) \to 0} \qquad \text{ERR IN}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P' \qquad P'_{ph} < 0}{A \vdash P \to 0}$$

$$\text{ERR SUM}: \frac{A \vdash P + Q \xrightarrow{\ell} P' \qquad P'_{ph} < 0}{A \vdash P + Q \to 0} \qquad \text{ERR TAU}: \frac{A \vdash \tau.P \xrightarrow{\tau} P \qquad P_{ph} < 0}{A \vdash \tau.P \to 0}$$

$$\text{ERR PAR}: \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q \qquad P'_{ph} < 0}{A \vdash P|Q \to Q} \qquad \text{ERR COMM}: \frac{A \vdash P|Q \xrightarrow{\tau} P' \qquad P'_{ph} < 0}{A \vdash P|Q \to 0}$$

$$\text{ERR RHO}: \frac{A \vdash *q \xrightarrow{\rho} Q \qquad Q_{ph} < 0}{A \vdash *q \to 0}$$

The rules above require that any process halt if its phlogiston balance is insufficient to cover the cost of the next transition. With respect to $\mathrm{ERR\ LIFT}$, notice the distinction between phlogiston balance of the null process 0_{ph} and the null process itself, also (unfortunately) denoted by 0. Also, the addition sign in $\mathrm{ERR\ SUM}$ once again represents non-deterministic choice, not addition as in the rules from the previous section.

With respect to $\mathrm{ERR\ PAR}$, notice that if P and Q can proceed independently, and $P'_{ph} < 0$, the execution of Q is unaffected. Intuitively, it makes sense that phlogiston balance of one process has no bearing on another, independent, process.

Future publications will define the more imperative programming constructs of Rholang, i.e., arithmetic expressions, case statements, etc. in a similar way. These are relatively straightforward to define. Additionally, future formulations will include type annotations and quantification of memory, storage, and network activity in the cost model.

Conclusion

Taking a step back we can see just how contracts are really just syntactic sugar for processes, and processes are made of processes. The compositionality we see in the natural world, this principle of “as above, so below” that reflects the tree in the structure of the leaf, that’s evident in the structure of social smart contracts like the ones being developed by startups like the Resonate Coop; and Nature’s trick of letting go of central authority to autonomy and independence, that’s also evident in the structure of social smart contracts on the RChain platform. That’s just what it means when we say that a process is the concurrent execution of several processes.