RCHAIN
COOPERATIVE

Rholang V 1.1

Lucius Gregory Meredith

RChain Research

RCHAIN
COOPERATIVE

Rholang V1.1

New and improved for-comprehension

for(

$ptrn_{11} \leftarrow src_{11}$ & … & $ptrn_{1n} \leftarrow src_{1n}$;

…

$ptrn_{m1} \leftarrow src_{m1}$ & … & $ptrn_{mn} \leftarrow src_{mn}$;

){P}

where src ::= x | x?! | x!?(a_1, …, a_k)

and ‘&’ replaces the old meaning of ‘;’

RCHAIN
COOPERATIVE

Rholang V1.1

Intuitively,

```
for( ptrn ← x?! ){P}
```

means wait on x for a tuple, of the form (v, r), where v is a value that will be pattern-matched against ptrn, and r is a return channel where an acknowledgement of the receipt of the value will be sent in parallel with P.

```
⟦for( ptrn ← x?! ){P}⟧  = for( ( ptrn, r ) ← x ){ r!() | ⟦P⟧ }
```

Thus, the decorations, x?!, are mnemonics for the fact that the expression waits (?) (on x) and then sends (!) (on the return channel r).

RCHAIN
COOPERATIVE

Rholang V1.1

Intuitively,

for(ptrn ← x!?(a_1, ..., a_k)){P}

means send on x the augmented argument list (a_1, ..., a_k, r) and then wait on r for a response which will be pattern-matched against ptrn before executing the continuation, P.

⟦for(ptrn ← x!?(a_1, ..., a_k)){P}⟧ = new r in { x!(a_1, ..., a_k, *r) | for(ptrn ← r){ ⟦P⟧ } }

Thus, the decorations, x!?, are mnemonics for the fact that the expression sends (!) (on x) and then waits (?) (on the return channel r).

RCHAIN
COOPERATIVE

Rholang V1.1

New and improved for-comprehension desugared

$$
\begin{aligned}
&[\![\text{for}(\\
&\quad ptrn_{11} \leftarrow x_{11}!?(a_1, \ldots, a_k) \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n};\\
&\qquad\qquad \ldots\\
&\quad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn};\\
&)\{P\}]\!]\\
&=\\
&\text{new } r_{11} \text{ in}\\
&\quad x_{11}!(a_1, \ldots, a_k, *r_{11})\\
&\quad \mid [\![\text{for}(\, ptrn_{11} \leftarrow r_{11} \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n};\\
&\qquad\qquad \ldots\\
&\qquad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn};\\
&\quad)\{P\}]\!]
\end{aligned}
$$

removing send/recv's: $x_{11}!?(a_1, \ldots, a_k)$

RCHAIN
COOPERATIVE

Rholang V1.1

New and improved for-comprehension desugared

$$
\begin{aligned}
&[\![\text{for}(\\
&\quad ptrn_{11} \leftarrow x_{11}?! \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n};\\
&\qquad\qquad \ldots\\
&\quad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn};\\
&)\{P\}]\!]\\
&=\\
&[\![\text{for}(\\
&\quad (ptrn_{11}, r) \leftarrow x_{11} \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n};\\
&\qquad\qquad \ldots\\
&\quad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn};\\
&)\{\, r!() \mid P\}]\!]
\end{aligned}
$$

removing recv/send's: $x_{11}?!$

where r is fresh for the whole context

RChain Research

Rholang V1.1

New and improved for-comprehension desugared

$$
\begin{aligned}
&[\![\text{for}(\\
&\quad ptrn_{11} \leftarrow x_{11} \,\&\, \ldots \,\&\, ptrn_{1n} \leftarrow x_{1n};\\
&\quad ptrn_{m1} \leftarrow src_{m1} \,\&\, \ldots \,\&\, ptrn_{mn} \leftarrow src_{mn};\\
&\qquad\qquad \ldots\\
&\quad ptrn_{m1} \leftarrow src_{m1} \,\&\, \ldots \,\&\, ptrn_{mn} \leftarrow src_{mn}\\
&)\{P\}]\!]\\
&=\\
&\text{for}(\\
&\quad ptrn_{11} \leftarrow x_{11} \,\&\, \ldots \,\&\, ptrn_{1n} \leftarrow x_{1n}\\
&)\{\\
&\quad [\![\text{for}(\\
&\qquad ptrn_{m1} \leftarrow src_{m1} \,\&\, \ldots \,\&\, ptrn_{mn} \leftarrow src_{mn};\\
&\qquad\qquad \ldots\\
&\qquad ptrn_{m1} \leftarrow src_{m1} \,\&\, \ldots \,\&\, ptrn_{mn} \leftarrow src_{mn};\\
&\quad)\{P\}]\!]\\
&\}
\end{aligned}
$$

removing ;’s

RCHAIN
COOPERATIVE

Rholang V1.1

sequential output

x!?(v);P

x!?(v).

Allows for sequences of sends

Intuitively, x!?(v);P sends the tuple, (v,*r) , on and then waits for an acknowledgement on r before running the continuation P.

RChain Research

RCHAIN
COOPERATIVE

Rholang V1.1

sequential send expressions desugared

⟦x!?(v);P⟧ = new r in x!((v,*r)) | for(_ ← r){ ⟦P⟧ }

removing ;'s

⟦x!?(v).⟧ = ⟦x!?(v);0⟧

removing .'s

RChain Research

RCHAIN
COOPERATIVE

Rholang V1.1

An example calculation

```
[[for( m ← x?!; n ← y?! ){ stdout!( “m+n = ” + *m + *n ) } | x!?( 1 ); y!?( 2 ).]]
=
for( (m,r1) ← x ){ r1!() | for( (n,r2) ← y ){ r2!() | stdout!( “m+n = ” + *m + *n ) }
| new r1 in { x!( (1,*r1) ) | for( _ ← r1 ){ new r2 in {  y!( (2,*r2) ) | for( _ ← r2 ){ 0 } } } }
=
for( (m,r1) ← x ){ r1!() | for( (n,r2) ← y ){ r2!() | stdout!( “m+n = ” + *m + *n ) }
| new r1 r2 in { x!( (1,*r1) ) | for( _ ← r1 ){ y!( (2,*r2) ) | for( _ ← r2 ){ 0 } } }
=
new r1 r2 in {
  for( (m,r1) ← x ){ r1!() | for( (n,r2) ← y ){ r2!() | stdout!( “m+n = ” + *m + *n ) }
  | x!( (1,*r1) ) | for( _ ← r1 ){ y!( (2,*r2) ) | for( _ ← r2 ){ 0 } }
}
```

RCHAIN COOPERATIVE

Rholang V1.1

An example calculation

→ comm(x,[@1/m,@*r_1/r_1])

```
new r1 r2 in {
  r1!() | for( (n,r2) ← y ){ r2!() | stdout!( “m+n = ” + 1 + n ) }
  | for( _ ← r1 ){ y!( (2,r2) ) | for( _ ← r2 ){ 0 } }
}
```

→ comm(r_1,[])

```
new r1 r2 in {
  for( (n,r2) ← y ){ r2!() | stdout!( “m+n = ” + 1 + n ) }
  | y!( (2,r2) ) | for( _ ← r2 ){ 0 }
}
```

Rholang V1.1

An example calculation

→ comm(y,[@2/n,@*r_2/r_2])

new r_1 r_2 in {

  r_2!() | stdout!(“m+n = ” + 1 + 2) | for(_ ← r_2){ 0 }

}

→ comm(r_2,[])

new r_1 r_2 in {

  stdout!(“m+n = ” + 1 + 2) | 0

}

=

stdout!(“m+n = ” + 1 + 2)

Guarantees the event log: (if we filter out all communications on unforgeable names)

comm(x,[@1/m,@*r_1/r_1]) < comm(y,[@2/n,@*r_2/r_2])

RCHAIN
COOPERATIVE

Rholang V1.1

An example calculation

⟦for(m ← x?!; n ← y?!){ stdout!(“m+n = ” + *m + *n) } | x!?(1); y!?(2).⟧
=
for((m,r_1) ← x){ r_1!() | for((n,r_2) ← y){ r_2!() | stdout!(“m+n = ” + *m + *n) }
| new r_1 in { x!((1,*r_1)) | for(_ ← r_1){ new r_2 in { y!((2,*r_2)) | for(_ ← r_2){ 0 } } } }

This example illustrates a better than 2X compression without loss of any of the rholang features.

RCHAIN
COOPERATIVE

Rholang V1.1

Taken together the improved for-comprehension and the sequential output set the stage for better performance.

$$
\begin{array}{l}
\text{for(} \\
\quad \text{ptrn}_{11} \leftarrow \text{src}_{11} \;\&\; \ldots \;\&\; \text{ptrn}_{1n} \leftarrow \text{src}_{1n}; \\
\quad \ldots \\
\quad \text{ptrn}_{m1} \leftarrow \text{src}_{m1} \;\&\; \ldots \;\&\; \text{ptrn}_{mn} \leftarrow \text{src}_{mn}; \\
\text{)\{P\}}
\end{array}
$$

$$
\begin{array}{l}
\text{x!?(v);P} \\
\\
\text{x!?(v).}
\end{array}
$$

The internal coordination which has no observable transactional import can be all be done, in principle, without hitting the tuple space.

Thus, rather than merely desugaring, we are proposing a compilation scheme. This will dramatically speed up rholang execution, in addition to providing a dramatic compression in code.

RChain Research

Rholang V1.1

let expressions

$$\text{let } ptrn_1 \leftarrow v_1 \text{ ; ... ; } ptrn_m \leftarrow v_m \text{ in P}$$

$$\text{let } ptrn_1 \leftarrow v_1 \text{ \& ... \& } ptrn_m \leftarrow v_m \text{ in P}$$

These provide immutable variables much like Scala's

```
val x = v; P
```

RCHAIN
COOPERATIVE

Rholang V1.1

let expressions desugared

$[\![\text{let } ptrn_1 \leftarrow v_1 ; \ldots ; ptrn_n \leftarrow v_n \text{ in } P]\!]$

=

```
new x1 in
  x1!(v1)
  | for( ptrn1 ← x1 ){
    [[let ptrn2 ← v2 ; … ; ptrnn ← vn in P ]]
  }
```

removing ;'s

RCHAIN
COOPERATIVE

Rholang V1.1

let expressions desugared

$[\![\text{let } ptrn_1 \leftarrow v_1 \;\&\; \ldots \;\&\; ptrn_n \leftarrow v_n \text{ in } P \,]\!]$

=

new $x_1 \ldots x_n$ in

$x_1!(v_1) \mid \ldots \mid x_n!(v_n)$

$\mid$ for($ptrn_1 \leftarrow x_1 \;\&\; \ldots \;\&\; ptrn_n \leftarrow x_n$){ $[\![P]\!]$ }

removing &'s

RCHAIN
COOPERATIVE

Rholang V1.1

In the case of sequential let the code compression is considerable; yet, the let expressions are designed not only for code compression

$$\text{let } ptrn_1 \leftarrow v_1 \text{ ; ... ; } ptrn_m \leftarrow v_m \text{ in P}$$

$$\text{let } ptrn_1 \leftarrow v_1 \text{ \& ... \& } ptrn_m \leftarrow v_m \text{ in P}$$

But provide an opportunity for a compilation scheme that dramatically speeds up rholang execution, because internal coordination communications need never hit the tuple space.