Rholang V1.1

New and improved for-comprehension

$$
\begin{aligned}
&\text{for(} \\
&\qquad ptrn_{11} \leftarrow src_{11} \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n}; \\
&\qquad\qquad\qquad \ldots \\
&\qquad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn}; \\
&\text{)\{P\}}
\end{aligned}
$$

where $src ::= x \mid x?! \mid x!?(\, a_1, \ldots, a_k \,)$

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

Intuitively,

$$\text{for(ptrn} \leftarrow \text{x?!)\{P\}}$$

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.

$$[\![\text{for(ptrn} \leftarrow \text{x?!)\{P\}}]\!] = \text{for((ptrn, r)} \leftarrow \text{x)\{ r!()} \mid [\![\text{P}]\!] \text{ \}}$$

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

Intuitively,

$$\text{for}(\ ptrn \leftarrow x!?(\ a_1, \ldots, a_k\ )\ )\{P\}$$

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

$$[\![\text{for}(\ ptrn \leftarrow x!?(\ a_1, \ldots, a_k\ ))\{P\}]\!] = \text{new } r \text{ in } \{\ x!(\ a_1, \ldots, a_k, {}^*r) \mid \text{for}(\ ptrn \leftarrow r\ )\{\ [\![P]\!]\ \}\ \}$$

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

New and improved for-comprehension desugared

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

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

New and improved for-comprehension desugared

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

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

where r is fresh for the whole context

New and improved for-comprehension desugared

$[\![\mathrm{for}($

$\quad ptrn_{11} \leftarrow x_{11}\ \&\ \ldots\ \&\ ptrn_{1n} \leftarrow x_{1n};$

$\quad ptrn_{m1} \leftarrow src_{m1}\ \&\ \ldots\ \&\ ptrn_{mn} \leftarrow src_{mn};$

$\quad \ldots$

$\quad ptrn_{m1} \leftarrow src_{m1}\ \&\ \ldots\ \&\ ptrn_{mn} \leftarrow src_{mn}$

$)\{P\}]\!]$

$=$

$\mathrm{for}($

$\quad ptrn_{11} \leftarrow x_{11}\ \&\ \ldots\ \&\ ptrn_{1n} \leftarrow x_{1n}$

$)\{$

$\quad [\![\mathrm{for}($

$\qquad ptrn_{m1} \leftarrow src_{m1}\ \&\ \ldots\ \&\ ptrn_{mn} \leftarrow src_{mn};$

$\qquad \ldots$

$\qquad ptrn_{m1} \leftarrow src_{m1}\ \&\ \ldots\ \&\ ptrn_{mn} \leftarrow src_{mn};$

$\quad)\{P\}]\!]$

$\}$

removing ;'s

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.

sequential send expressions desugared

$$[\![x!?(v);P]\!] = \text{new } r \text{ in } x!((v,{}^*r)) \mid \text{for}(\ \_ \leftarrow r\ )\{\ [\![P]\!]\ \}$$

removing ;'s

$$[\![x!?(v).]\!] = [\![x!?(v);0]\!]$$

removing .'s

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 } } } }

=

for((m,r_1) ← x){ r_1!() | for((n,r_2) ← y){ r_2!() | stdout!(“m+n = ” + *m + *n) }

| new r_1 r_2 in { x!((1,*r_1)) | for(_ ← r_1){ y!((2,*r_2)) | for(_ ← r_2){ 0 } } }

=

new r_1 r_2 in {

  for((m,r_1) ← x){ r_1!() | for((n,r_2) ← y){ r_2!() | stdout!(“m+n = ” + *m + *n) }

  | x!((1,*r_1)) | for(_ ← r_1){ y!((2,*r_2)) | for(_ ← r_2){ 0 } }

}

An example calculation

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

An example calculation

$\rightarrow$ comm(y,[@2/n,@*r_2/r_2])

new r_1 r_2 in {

  r_2!() | stdout!("m+n = " + 1 + 2) | for(_ $\leftarrow$ r_2){ 0 }

}

$\rightarrow$ 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])

An example calculation

$$[\![\mathrm{for}(\ m \leftarrow x?!;\ n \leftarrow y?!\ )\{\ \mathrm{stdout}!(\ \text{“m+n = ”} + {}^*m + {}^*n\ )\ \} \mid x!?(\ 1\ );\ y!?(\ 2\ ).]\!]$$

$$=$$

$$\mathrm{for}(\ (m,r_1) \leftarrow x\ )\{\ r_1!() \mid \mathrm{for}(\ (n,r_2) \leftarrow y\ )\{\ r_2!() \mid \mathrm{stdout}!(\ \text{“m+n = ”} + {}^*m + {}^*n\ )\ \}$$

$$\mid \mathrm{new}\ r_1\ \mathrm{in}\ \{\ x!(\ (1,{}^*r_1)\ ) \mid \mathrm{for}(\ \_ \leftarrow r_1\ )\{\ \mathrm{new}\ r_2\ \mathrm{in}\ \{\ \ y!(\ (2,{}^*r_2)\ ) \mid \mathrm{for}(\ \_ \leftarrow r_2\ )\{\ 0\ \}\ \}\ \}\ \}$$

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

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

$$
\begin{array}{l}
\text{for(} \\
\qquad ptrn_{11} \leftarrow src_{11} \;\&\; \ldots \;\&\; ptrn_{1n} \leftarrow src_{1n}; \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad ptrn_{m1} \leftarrow src_{m1} \;\&\; \ldots \;\&\; ptrn_{mn} \leftarrow src_{mn}; \\
\text{)\{P\}}
\end{array}
$$

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

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.

let expressions

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

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

These provide immutable variables much like Scala's

val x = v; P

let expressions desugared

$$
\begin{aligned}
&[\![\text{let } ptrn_1 \leftarrow v_1 \,;\, \ldots \,;\, ptrn_n \leftarrow v_n \text{ in } P]\!] \\
&= \\
&\text{new } x_1 \text{ in} \\
&\quad x_1!(v_1) \\
&\quad \mid \text{for}(\, ptrn_1 \leftarrow x_1 \,)\{ \\
&\qquad [\![\text{let } ptrn_2 \leftarrow v_2 \,;\, \ldots \,;\, ptrn_n \leftarrow v_n \text{ in } P]\!] \\
&\quad \}
\end{aligned}
$$

removing ;'s

let expressions desugared

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

$$=$$

$$\text{new } x_1 \ldots x_n \text{ in}$$

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

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

removing &'s

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

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

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

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