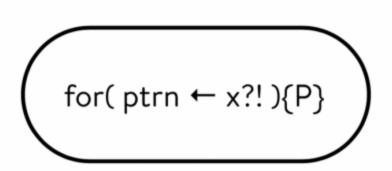
# Rholang V1.1

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 \mid x?! | x!?( a_1, ..., a_k ) and '&' replaces the old meaning of ';'
```

# Intuitively,



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.

$$\left[ [for(ptrn \leftarrow x?!)\{P\}] = for((ptrn, r) \leftarrow x)\{r!()|[P]] \} \right]$$

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

#### Intuitively,

means send on x the augmented argument list ( a<sub>1</sub>, ..., a<sub>k</sub>, r ) and then wait on r for a response which will be pattern-matched against ptrn before executing the continuation, P.

```
\left( [for(ptrn \leftarrow x!?(a_1, ..., a_k)){P}] = new r in \{ x!(a_1, ..., a_k, *r) | for(ptrn \leftarrow r){[P]} \} \right)
```

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

```
[for(
       ptrn_{11} \leftarrow x_{11}!?(a_1, ..., a_k) \& ... \& ptrn_{1n} \leftarrow src_{1n};
       ptrn_{m1} \leftarrow src_{m1} \& ... \& ptrn_{mn} \leftarrow src_{mn};
){P}]
new r<sub>11</sub> in
 x_{11}!(a_1, ..., a_{l}, *r_{11})
  | [for(ptrn_{11} \leftarrow r_{11} \& ... \& ptrn_{1n} \leftarrow src_{1n};
       ptrn_{m1} \leftarrow src_{m1} \& ... \& ptrn_{mn} \leftarrow src_{mn};
      ){P}]
```

removing send/recv's: x<sub>11</sub>!?( a<sub>1</sub>, ..., a<sub>k</sub> )

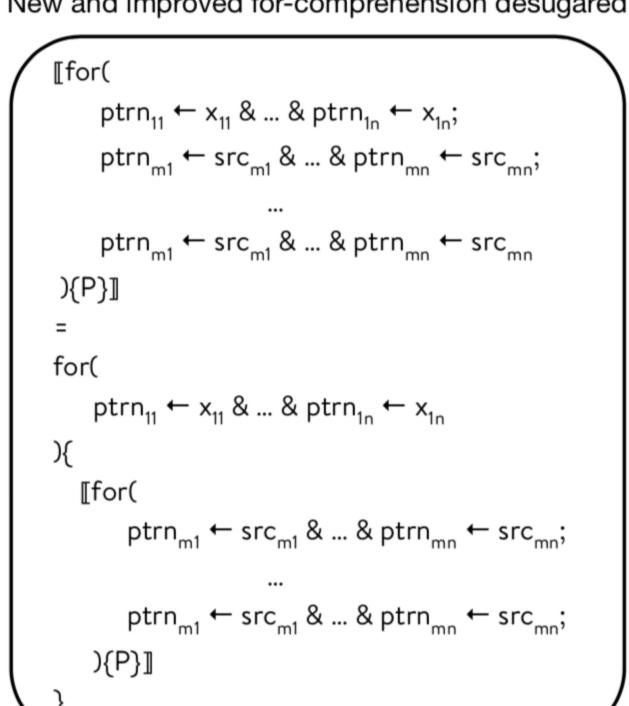
New and improved for-comprehension desugared

```
[for(
       ptrn_{11} \leftarrow x_{11}?! \& ... \& ptrn_{1n} \leftarrow src_{1n};
       ptrn_{m1} \leftarrow src_{m1} \& ... \& ptrn_{mn} \leftarrow src_{mn};
){P}]
[for(
       (ptrn_{11}, r) \leftarrow x_{11} \& ... \& ptrn_{1n} \leftarrow src_{1n};
       ptrn_{m1} \leftarrow src_{m1} \& ... \& ptrn_{mn} \leftarrow src_{mn};
      ){ r!() | P}]
```

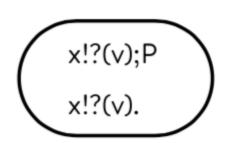
removing recv/send's: x<sub>11</sub>?!

where r is fresh for the whole context

New and improved for-comprehension desugared



## sequential output



## 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

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

removing;'s

#### An example calculation

```
[for( m \leftarrow x?!; n \leftarrow y?!){ stdout!("m+n = " + *m + *n)}|x!?(1); y!?(2).]
for( (m,r_1) \leftarrow x){ r_1!() | for( (n,r_2) \leftarrow y){ r_2!() | stdout!( "m+n = " + *m + *n ) }
 new r_1 in \{x!((1,*r_1)) | for(_ \leftarrow r_1) \{ new r_2 in \{ y!((2,*r_2)) | for(_ \leftarrow r_2) \{ 0 \} \} \} \}
for( (m,r_1) \leftarrow x){ r_1!() | for( (n,r_2) \leftarrow y){ r_2!() | stdout!( "m+n = " + *m + *n ) }
 new r_1 r_2 in \{ x!((1,*r_1)) | for(_ \leftarrow r_1) \{ y!((2,*r_2)) | for(_ \leftarrow r_2) \{ 0 \} \} \}
new r_1 r_2 in {
  for((m,r_1) \leftarrow x){r_1!()| for((n,r_2) \leftarrow y){r_2!()| stdout!("m+n = " + *m + *n)}
  |x!((1,*r_1))| for ( - \leftarrow r_1) \{ y!((2,*r_2)) | for ( - \leftarrow r_2) \{ 0 \} \}
```

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

 $|y!((2,r_2))|$  for  $(-\leftarrow r_2)\{0\}$ 

 $\rightarrow$  comm(r<sub>1</sub>,[])

new  $r_1 r_2$  in {

## 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(_ \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)

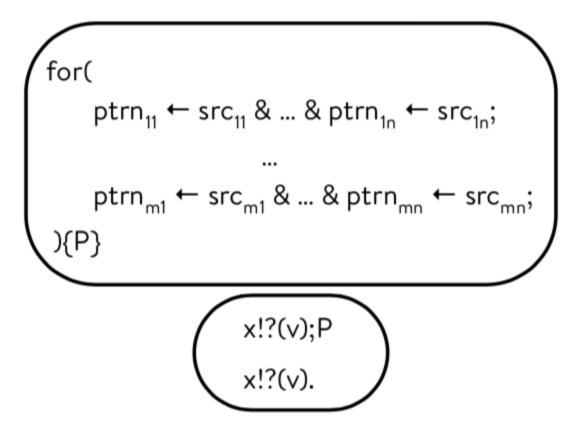
$$comm(x,[@1/m,@*r_1/r_1]) < comm(y,[@2/n,@*r_2/r_2])$$

## An example calculation

```
[for( m \leftarrow x?!; n \leftarrow y?! ){ stdout!( "m+n = " + *m + *n ) } | x!?(1); y!?(2).]
\begin{split} &\text{for( (m,r_1) \leftarrow x )\{ \ r_1!0 \ | \ \text{for( (n,r_2) \leftarrow y )\{ \ r_2!0 \ | \ \text{stdout!( "m+n = " + *m + *n ) }\} \\ &\text{| new } \ r_1 \ \text{in } \{ \ x!( \ (1,^*r_1) \ ) \ | \ \text{for( \_ \leftarrow r_1)\{ new } \ r_2 \ \text{in } \{ \ y!( \ (2,^*r_2) \ ) \ | \ \text{for( \_ \leftarrow r_2)\{ 0 \} \} } \} \\ \end{split}
```

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.



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

let 
$$ptrn_1 \leftarrow v_1; ...; ptrn_m \leftarrow v_m in P$$
  
let  $ptrn_1 \leftarrow v_1 \& ... \& ptrn_m \leftarrow v_m in P$ 

These provide immutable variables much like Scala's

val 
$$x = v$$
; P

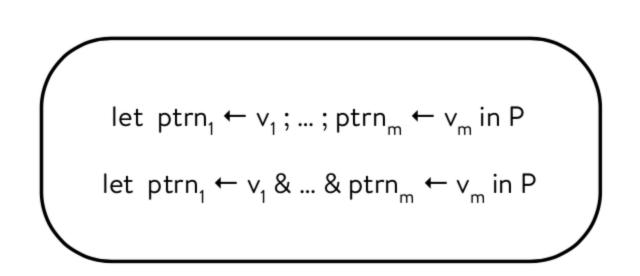
## let expressions desugared

```
[[\text{let ptrn}_1 \leftarrow v_1; ...; \text{ptrn}_n \leftarrow v_n \text{ in P}]]
new x₁ in
   x_1!(v_1)
   | for( ptrn_1 \leftarrow x_1) |
          [let ptrn<sub>2</sub> \leftarrow v<sub>2</sub>; ...; ptrn<sub>n</sub> \leftarrow v<sub>n</sub> in P]
```

## let expressions desugared

removing &'s

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



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