Termination Semantics

Joseph Denman

RChain Cooperative

Boulder Developer Conference

18 April 2018

Termination Model

Rholang can construct infinitely reducing programs.

- An actor may deploy infinitely reducing programs at no cost.

Need to constrain the language to only produce finite (terminating) *processes.*

- A Rholang program has “terminated” when it ceases to cause effects.

Solution: An actor must pay the miner to construct a termination proof.

Termination Proof

Termination argument search (*Turing, 1949*): a ranking function $\mathbf{\Phi : \Sigma \rightarrow W}$ maps each program state to an element of a well-order (**W**).

- Any well-order will do, but the order must be universal across a validator set.

Termination argument checking (*Turing, 1949*): a transition invariant $P \rightarrow P' \wedge \mathbf{\Phi}(P) \geq \min \mathbf{W} \Rightarrow \mathbf{\Phi}(P) > \mathbf{\Phi}(P')$ verifies that the well-order value is decreasing.

- Evaluating the program provides a termination proof.
- If termination argument check is false, the program is a counter-example; it must fail.

Termination operation search: a ranking function $\mathbf{\Omega : L \rightarrow W}$ maps each program operation to an element of a well-order(**W**).

- $\mathbf{\Phi}(P) > \mathbf{\Phi}(P) - \mathbf{\Omega}(\rightarrow) = \mathbf{\Phi}(P')$.
- $\mathbf{\Omega}$ is tuned to reflect the computational complexity or "cost" of $\rightarrow$.

Economic Incentives

Evaluating a program = constructing a termination proof.

- So far, evaluation only consumes miner resources.
- Long-running computations are still free for actor, so there's no disincentive for DoS.

"... just DoS the proof." – Bad Actor

Evaluating a program should consume actor's resources as well, so:

1. Actor specifies a $\mathbf{W}_{max}$
2. Actor specifies a conversion rate: **Rev** / **W**
3. Actor purchases $\mathbf{W}_{max}$ at the conversion rate: $\mathbf{W}_{max}$* (**Rev** / **W**) = **Rev**,
4. **Rev** is removed from actor's account
5. $\mathbf{W}_{max} = \mathbf{\Phi}(P_{init}) = ph_{init}$
6. If proof fails, **Rev** is yielded to miner and all effects of the program are reverted.

Phlo may be distributed to the sub-terms of a process.

$$\frac{(P \mid Q, ph)}{(P, ph) \mid (Q, ph)}$$

Phlo modifications can happen concurrently.

$$\frac{(P, ph) \to (P', ph') \qquad\qquad ph' \geq 0}{(P, ph) \mid (Q, ph) \to (P', ph') \mid (Q, ph')}$$

A process halts if it tries to yield a negative phlo balance.

$$\frac{(P, ph) \to (P', ph') \qquad\qquad ph' < 0}{(P, ph) \to (Nil, ph)}$$

```
   Deploy(1+1 | 1+1, 2, …)

=  S | (1+1 | 1+1, ph = 2)

=> S | (1+1, ph = 2) | (1+1, ph = 2)

=> S | (2, ph = 1) | (1+1, ph = 1)

=> S | (2, ph = 0) | (2, ph = 0)
```

```
   Deploy(1+1 | 1+1 | 1+1, 2, …)

=  S | (1+1 | 1+1 | 1+1, ph = 2)

=> S | (1+1, ph = 2) | (1+1, ph = 2) | (1+1, ph = 2)

=> S | (2, ph = 1) | (1+1, ph = 1) | (1+1, ph = 1)

=> S | (2, ph = 0) | (2, ph = 0) | (1+1, ph = 0)

=> S | (2, ph = 0) | (2, ph = 0) | (Nil, ph = 0)

=> OutOfPhloError(OOPE)
```

Tuplespace Grammar

```
C  := [S_1, …, S_N]

S  := (!@Q, ph)            // Ephemeral Write
      (!!@Q, ph)           // Persistent Write
      (?z.P, ph)           // Ephemeral Read
      (??z.P, ph)          // Persistent Read

T  := {x_1 -> C_1, …, x_n -> C_n}
```

```
P := @“dataFeed”!!(“data”) | for(z <- @“dataFeed”){*z}

   Deploy(P,…)

=  S | (@“dataFeed”!!(“data”) | for(z <- @“dataFeed”){*z}, ph)

   T := { @“dataFeed” -> [] }

=> S | (@“dataFeed”!!(“data”), ph) | (for(z <- @“dataFeed”){*z}, ph)

   T := { @“dataFeed” -> [] }

=> S | (for(z <- @“dataFeed”){*z}, ph’)

   T := { @“dataFeed” -> [(!!@“data”, ph’)] }
```

```
=> S | (*@“data”, ph’’’)

   T := { @“dataFeed” -> [(!!@“data”, ph’’’)] }

=> S | (“data”, ph’’’’)

   T := { @“dataFeed” -> [(!!@“data”, ph’’’’)] }
```

```
P := @“dataFeed”!!(“data”)

Q := for(z <- @“dataFeed”){*z}

   Deploy(P, …) | Deploy(Q, …)

=  S | (@“dataFeed”!!(“data”), ph_A) | (for(z <- @“dataFeed”){*z}, ph_B)

   T := { @“dataFeed” -> [] }

=> S | (@“dataFeed”!!(“data), ph_A)

   T := { @“dataFeed” -> [(?z.*z, ph_B’)] }
```

```
=> S | (*@“data”, ph_B’)

   T := { @“dataFeed” -> [(!!@“data”, ph_A’’)] }

=> S | (“data”, ph_B’’)

   T := { @“dataFeed” -> [(!!@“data”, ph_A’’)] }
```

Contracts

The only semantic difference between a persistent receive and a contract definition is in who pays for the continuation.

```
contract X(z)={P} := for(z <= X){P}
            X(Q) := X!(Q)
```

The left indicates that the *invoker* will pay for `P; X(Q)` links phlo supply to `P`.

- A useful distinction when publishing processes.
- Deploying a contract *definition* only requires payment for the first “receive”.

The right indicates that the *deployer* will pay for `P; X!(Q)` does not link phlo supply to `P`.

```
P := contract X(z)={P}

Q := for(z <= X){P},

   Deploy(P, …) | Deploy(Q, …)

=> S | (contract X(z)={P}, ph_A) | (for(z <= X){P}, ph_B)

   T := { X -> []}

=> S | (contract X(z)={P}, ph_A)

   T := { X -> [(??z.P, ph_B’)] }

=> S

   T := { X -> [(??z.P, ph_B’),(??z.P, ⊥)] }
```

=> S := S′ | (X(Q), ph_C) | (X!(Q), ph_D)

T := { X -> [(??z.P, ph_B′),(??z.P, ⊥)] }

=> S′ | (P{@Q/z}, ph_B′′) | (P{@Q/z}, ph_C′)

T := { X -> [(??z.P, ph_B′′),(??z.P, ⊥)] }

where (X(Q), ph_C) -> (Nil, ph_C′)

and (X!(Q), ph_D) -> (Nil, ph_D′)