The Aneris Documentation

March 15, 2023

Abstract

This document formally describes the Aneris program logic. The latest version of this document and the Coq formalization can be found in the git repository at https://github.com/logsem/aneris.

Contents

1 Language

1.1 Abstract Syntax

AnerisLang is an untyped functional language with higher-order functions, fork-based concurrency, higher-order mutable references, and primitives for communicating over network sockets. The abstract syntax is as follows.

$$
\begin{aligned}
z &\in \mathbb{Z} \\
s &\in \mathit{String} \\
x \in \mathit{Var},\ \mathit{sh} \in \mathit{Handle},\ \ell \in \mathit{Loc} &\triangleq (\mathit{infinite\ countable\ set}) \\
l \in \mathit{BaseLit} &::= z \mid \texttt{true} \mid \texttt{false} \mid () \mid \ell \mid s \mid \mathit{sh} \\
v \in \mathit{Val} &::= l \mid \texttt{rec}\ f\ x = e \mid (v_1, v_2) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\
\circledcirc_1 &::= \sim \mid - \mid \texttt{i2s} \mid \texttt{s2i} \mid \texttt{len} \\
\circledcirc_2 &::= + \mid - \mid * \mid \texttt{quot} \mid \texttt{rem} \mid \& \mid \mid \mid \hat{} \mid << \mid >> \mid \leq \mid < \mid = \mid ++ \\
e \in \mathit{Expr} &::= v \mid x \mid e_1\ e_2 \mid \circledcirc_1\, e \mid e_1 \circledcirc_2 e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
&\quad \mid \texttt{find}\ e_1\ e_2\ e_3 \mid \texttt{substring}\ e_1\ e_2\ e_3 \mid \texttt{rand}\ e \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
&\quad \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{match}\ e\ \texttt{with}\ \texttt{inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2\ \texttt{end} \\
&\quad \mid \texttt{fork}\ \{e\} \mid \texttt{ref}\ \ e \mid\ !e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \\
&\quad \mid \texttt{makeaddress}\ e_1\ e_2 \mid \texttt{getaddress}\ e \mid \texttt{socket}\ \mid \texttt{socketbind}\ e_1\ e_2 \\
&\quad \mid \texttt{sendto}\ e_1\ e_2\ e_3 \mid \texttt{receivefrom}\ e \mid \texttt{settimeout}\ e_1\ e_2\ e_3 \mid \texttt{start}\ l\ e
\end{aligned}
$$

We introduce the following syntactic sugar: lambda abstractions $\lambda x.\, e$ defined as $\texttt{rec}\ \_\ x = e$, let-bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ defined as $(\lambda x.\, e_2)(e_1)$, sequencing $e_1; e_2$ defined as $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$, assertions $\texttt{assert}\ e$ defined as $\texttt{if}\ e\ \texttt{then}\ ()\ \texttt{else}\ 0\ 0$, $\texttt{None}$ defined as $\texttt{inl}\ ()$ and $\texttt{Some}\ x$ defined as $\texttt{inr}\ x$.

We have the usual operations on locations $\ell \in \mathit{Loc}$ in the heap: $\texttt{ref}\ v$ for allocating a new reference, $!\,\ell$ for dereferencing, and $\ell \leftarrow v$ for assignment. $\texttt{CAS}\ \ell\ v_1\ v_2$ is an atomic compare-and-set operation used to achieve synchronization between threads on a specific memory location ℓ. Operationally, it tests whether ℓ has value v_1 and if so, updates the location to v_2, returning a boolean indicating whether the swap succeeded or not.

The binary operations $\&$, $|$, $\hat{}$, $<<$, $>>$ are all bitwise operations, respectively: and, or, exclusive or, left shift, right shift.

The operation **find** finds the index of a particular substring in a string $s \in \mathit{String}$ and **substring** splits a string at given indices, producing the corresponding substring. **i2s** and **s2i** convert between integers and strings. These operations are mainly used for serialization and deserialization purposes.

The expression $\texttt{fork}\ \{e\}$ forks off a new (node-local) thread and $\texttt{start}\ n\ e$ will spawn a new node $n \in \mathit{IP\text{-}Address}$ (we use ip addresses as node identifiers modeled on the set of strings) running the program e. Note that it is only the system node $\mathfrak{S}$ ($\mathfrak{S}$ is a particular ip address we use to represent the system node) who is able to spawn nodes. The intention is for this to be done during the bootstrapping phase of a distributed system.

We use $\mathit{sh} \in \mathit{Handle}$ to range over socket handles created by the **socket** operation. **makeaddress** constructs a socket address given an ip address and a port, and the network primitives **socketbind**, **sendto**, **receivefrom** and **settimeout** correspond to the similar BSD-socket API methods. Do note that in the Coq implementation, **socket** takes three values, in accordance with the similar BSD-socket API method, in form of an address family PF_INET, socket type SOCK_DGRAM and protocol type IPPROTO_UDP. As we do not support other types of address families, sockets or protocols, we omit these from this document.

1.2 Operational Semantics

The operational semantics is defined using the following evaluation contexts, where $\bullet$ denotes the empty evaluation context. Observe that the evaluation contexts imply right-to-left evaluation order.

$$
\begin{aligned}
K \in \mathit{Ctx} ::= {} & \bullet \mid K\ v \mid e\ K \mid \circledcirc_1\ K \mid K\ \circledcirc_2\ v \mid e\ \circledcirc_2\ K \mid \texttt{if}\ K\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{find}\ K\ v_1\ v_2 \\
& \mid \texttt{find}\ e\ K\ v \mid \texttt{find}\ e_1\ e_2\ K \mid \texttt{substring}\ K\ v_1\ v_2 \mid \texttt{substring}\ e\ K\ v \\
& \mid \texttt{substring}\ e_1\ e_2\ K \mid \texttt{rand}\ K \mid (K, v) \mid (e, K) \mid \texttt{fst}\ K \mid \texttt{snd}\ K \mid \texttt{inl}\ K \\
& \mid \texttt{inr}\ K \mid \texttt{match}\ K\ \texttt{with inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2\ \texttt{end} \mid \texttt{ref}\ K \mid\ !K \mid K \leftarrow v \\
& \mid e \leftarrow K \mid \texttt{CAS}\ K\ v_1\ v_2 \mid \texttt{CAS}\ e\ K\ v \mid \texttt{CAS}\ e_1\ e_2\ K \mid \texttt{makeaddress}\ K\ v \\
& \mid \texttt{makeaddress}\ e\ K \mid \texttt{getaddress}\ K \mid \texttt{socketbind}\ K\ v \mid \texttt{socketbind}\ e\ K \\
& \mid \texttt{sendto}\ K\ v_1\ v_2 \mid \texttt{sendto}\ e\ K\ v \mid \texttt{sendto}\ e_1\ e_2\ K \mid \texttt{settimeout}\ K\ v_1\ v_2 \\
& \mid \texttt{settimeout}\ e\ K\ v \mid \texttt{settimeout}\ e_1\ e_2\ K \mid \texttt{receivefrom}\ K
\end{aligned}
$$

In AnerisLang, the state is of the form $(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M})$ where $\mathcal{M}$ is a message soup, a multiset of messages in transit, $\vec{\mathcal{H}}$ and $\vec{\mathcal{S}}$ mappings from IP addresses to, respectively, node-local heaps $\mathcal{H}$ and node-local socket mappings $\mathcal{S}$. We model sockets as a pair where the first element is a socket address together with a blocking-bit, indicating whether the socket is blocking upon receiving messages, and the second element is a receive buffer. Formally a state is of the following type, where we use $\mathcal{P}^+$ to denote the power multiset:

$$
\begin{aligned}
\mathit{IP\text{-}Address} &\triangleq \mathit{String} \\
\mathsf{op}(A) &\triangleq \texttt{None} \mid \texttt{Some}\ a\ (a \in A) \\
\mathit{Port} &\triangleq \mathbb{N} \\
\mathit{Message} &\triangleq (\mathit{IP\text{-}Address} \times \mathit{Port}) \times (\mathit{IP\text{-}Address} \times \mathit{Port}) \times \mathit{String} \\
\mathit{NetworkState} &\triangleq \left\{ \begin{array}{l} \mathcal{S} : \mathit{Handle} \xrightarrow{\mathsf{fin}} ((\mathsf{op}(\mathit{IP\text{-}Address} \times \mathit{Port}) \times \mathbb{B}) \times \mathit{List}(\mathit{Message})), \\ \mathcal{M} : \mathcal{P}^+(\mathit{Message}) \end{array} \right\} \\
\mathit{State} &\triangleq \left\{ \begin{array}{l} \vec{\mathcal{H}} : \mathit{IP\text{-}Address} \xrightarrow{\mathsf{fin}} (\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}), \\ \vec{\mathcal{S}} : \mathit{IP\text{-}Address} \xrightarrow{\mathsf{fin}} (\mathit{Handle} \xrightarrow{\mathsf{fin}} ((\mathsf{op}(\mathit{IP\text{-}Address} \times \mathit{Port}) \times \mathbb{B}) \times \mathit{List}(\mathit{Message}))), \\ \mathcal{M} : \mathcal{P}^+(\mathit{Message}) \end{array} \right\}
\end{aligned}
$$

For notational convenience, we let m_{src}, m_{dst} and m_{str} denote respectively the first, second and third projection of the message m. We write ϵ for the empty list of messages. We proceed by introducing the stepping relations bottom up.

Socket-Step ($\rightsquigarrow_{(n)}$)

Socket-Step has the signature

$$\mathit{Expr} \times \mathit{NetworkState} \rightsquigarrow_{(n)} \mathit{Expr} \times \mathit{NetworkState}.$$

Sockets steps are node local network aware steps, i.e. steps happening at a particular node defined by its IP address that may use and modify the network-related state (sockets and message soup). The Socket-Step relation is parameterized by n, the ip of the node executing the step. Remark that we only model timeouts partially: Timeouts only decide whether a socket is blocking or not, hence we treat all non-zero timeouts the same.

NEW-SOCKET

$$\frac{sh \notin \mathsf{dom}(\mathcal{S})}{(\texttt{socket}\ (), (\mathcal{S}, \mathcal{M})) \rightsquigarrow_{(n)} (sh, (\mathcal{S}[sh \mapsto ((\texttt{None}, \texttt{true}), \epsilon)], \mathcal{M}))}$$

SOCKET-BIND

$$\frac{\forall sh' \in \mathsf{dom}(\mathcal{S}).\, \mathcal{S}(sh') = ((\texttt{Some}\ (n', p'), b'), \epsilon) \Rightarrow p' \neq p}{(\texttt{socketbind}\ sh\ (n, p), (\mathcal{S}[sh \mapsto ((\texttt{None}, b), \epsilon)], \mathcal{M})) \rightsquigarrow_{(n)} (0, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \epsilon)], \mathcal{M}))}$$

SEND-MESSAGE

$$(\texttt{sendto}\ sh\ str\ dst, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B})], \mathcal{M})) \rightsquigarrow_{(n)} (|str|, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B})], \mathcal{M} \uplus \{((n, p), dst, str)\}))$$

RECEIVE-SOME-MESSAGE

$$(\texttt{receivefrom}\ sh, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B} +\!\!+ [m])], \mathcal{M})) \rightsquigarrow_{(n)} (\texttt{Some}\,(m_{str}, m_{src}), (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B})], \mathcal{M}))$$

RECEIVE-EMPTY-BUFFER-NON-BLOCKING

$$(\texttt{receivefrom}\ sh, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{false}), \epsilon)], \mathcal{M})) \rightsquigarrow_{(n)} (\texttt{None}, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{false}), \epsilon)], \mathcal{M}))$$

RECEIVE-EMPTY-BUFFER-BLOCKING

$$(\texttt{receivefrom}\ sh, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{true}), \epsilon)], \mathcal{M})) \rightsquigarrow_{(n)} (\texttt{receivefrom}\ sh, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{true}), \epsilon)], \mathcal{M}))$$

SET-SOCKET-TO-NON-BLOCKING-RECEIVE

$$\frac{(0 \leq x \wedge 0 \leq y \wedge 0 < (x + y))}{(\texttt{settimeout}\ sh\ x\ y, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B})], \mathcal{M})) \rightsquigarrow_{(n)} ((), (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{false}), \mathcal{B})], \mathcal{M}))}$$

SET-SOCKET-TO-BLOCKING-RECEIVE

$$(\texttt{settimeout}\ sh\ 0\ 0, (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), b), \mathcal{B})], \mathcal{M})) \rightsquigarrow_{(n)} ((), (\mathcal{S}[sh \mapsto ((\texttt{Some}\ (n, p), \texttt{true}), \mathcal{B})], \mathcal{M}))$$

Base-Step ($\rightsquigarrow_b$)

Base-Step has the signature

$$\mathit{Expr} \times (\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}) \rightsquigarrow_b \mathit{Expr} \times (\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}) \times \mathit{List}(\mathit{Expr}).$$

Base-step is a non network aware step that may modify the node local heap. Fork allows for creation of new threads, under the same node as the current execution, hence the presence of a list of new expressions in the relation where ϵ denotes the empty list. Note the Get-Address-Info-Step is redundant in this representation but is included to reflect the BSD-socket API. We use the terminology pure about a reduction with no side effects. The b in $\rightsquigarrow_b$ is a part of the name of the relation and not a parameter.

Pure Reductions

BETA-STEP

$$((\texttt{rec}\, f(x) = e)(v), \mathcal{H}) \rightsquigarrow_b (e[(\texttt{rec}\, f(x) = e)/f][v/x], \mathcal{H}, \epsilon)$$

UNARY-OPERATION-STEP

$$(-_{\circledcirc} z, \mathcal{H}) \rightsquigarrow_b (-z, \mathcal{H}, \epsilon)$$

Binary-Operation-Step
$(z_1 +_{\circledcirc} z_2, \mathcal{H}) \rightsquigarrow_b (z_1 + z_2, \mathcal{H}, \epsilon)$

If-True-Step
$(\texttt{if true then } e_1 \texttt{ else } e_2, \mathcal{H}) \rightsquigarrow_b (e_1, \mathcal{H}, \epsilon)$

If-False-Step
$(\texttt{if false then } e_1 \texttt{ else } e_2, \mathcal{H}) \rightsquigarrow_b (e_2, \mathcal{H}, \epsilon)$

Find-From-Step
$$\frac{n \geq n_2 \qquad s_1 = s_3[n \cdots (n + n')]}{(\texttt{find } s_1 \ n_2 \ s_3, \mathcal{H}) \rightsquigarrow_b (n, \mathcal{H}, \epsilon)}$$

Substring-Step
$(\texttt{substring } s \ n_1 \ n_2, \mathcal{H}) \rightsquigarrow_b (s[n_1 \cdots (n_1 + n_2)], \mathcal{H}, \epsilon)$

Rand-Step
$$\frac{0 \leq r < u}{(\texttt{rand } u, \mathcal{H}) \rightsquigarrow_b (r, \mathcal{H}, \epsilon)}$$

Fst-Step
$(\texttt{fst } (v_1, v_2), \mathcal{H}) \rightsquigarrow_b (v_1, \mathcal{H}, \epsilon)$

Snd-Step
$(\texttt{snd } (v_1, v_2), \mathcal{H}) \rightsquigarrow_b (v_2, \mathcal{H}, \epsilon)$

Case-Left-Step
$(\texttt{match inl } v \texttt{ with inl } x_1 \Rightarrow e_1 \mid \texttt{inr } x_2 \Rightarrow e_2 \texttt{ end}, \mathcal{H}) \rightsquigarrow_b (e_1[v/x_1], \mathcal{H}, \epsilon)$

Case-Right-Step
$(\texttt{match inr } v \texttt{ with inl } x_1 \Rightarrow e_1 \mid \texttt{inr } x_2 \Rightarrow e_2 \texttt{ end}, \mathcal{H}) \rightsquigarrow_b (e_2[v/x_2], \mathcal{H}, \epsilon)$

Make-Address-Step
$(\texttt{makeaddress } n \ p, \mathcal{H}) \rightsquigarrow_b ((n, p), \mathcal{H}, \epsilon)$

Get-Address-Info-Step
$(\texttt{getaddress } (n, p), \mathcal{H}) \rightsquigarrow_b ((n, p), \mathcal{H}, \epsilon)$

Impure Reductions

Alloc-Step
$$\frac{\ell \notin \mathsf{dom}(\mathcal{H})}{(\texttt{ref } \ v, \mathcal{H}) \rightsquigarrow_b (\ell, \mathcal{H}[\ell \mapsto v], \epsilon)}$$

Load-Step
$(!\,\ell, \mathcal{H}[\ell \mapsto v]) \rightsquigarrow_b (v, \mathcal{H}[\ell \mapsto v], \epsilon)$

Store-Step
$(\ell \leftarrow v, \mathcal{H}[\ell \mapsto w]) \rightsquigarrow_b ((), \mathcal{H}[\ell \mapsto v], \epsilon)$

CAS-Fail-Step
$$\frac{v \neq v_1}{(\texttt{CAS } \ell \ v_1 \ v_2, \mathcal{H}[\ell \mapsto v]) \rightsquigarrow_b (\texttt{false}, \mathcal{H}[\ell \mapsto v], \epsilon)}$$

CAS-Success-Step
$(\texttt{CAS } \ell \ v_1 \ v_2, \mathcal{H}[\ell \mapsto v_1]) \rightsquigarrow_b (\texttt{true}, \mathcal{H}[\ell \mapsto v_2], \epsilon)$

Fork-Step
$(\texttt{fork } \{e\}, \mathcal{H}) \rightsquigarrow_b ((), \mathcal{H}, [e])$

Head-Step ($\rightsquigarrow$)

Head-Step has the signature

$$(\textit{IP-Address} \times \textit{Expr}) \times \textit{State} \rightsquigarrow (\textit{IP-Address} \times \textit{Expr}) \times \textit{State} \times \textit{List}(\textit{IP-Address} \times \textit{Expr}).$$

Head-Step is aware of the whole state and is defined using the socket-step and base-step. Importantly the head-step allows for creation of new nodes through the Assign-New-Ip-Step rule. Note only the system node $\mathfrak{S}$ is able to create new nodes.

$$\textsc{Local-Pure-Step}\quad \frac{(e, \mathcal{H}) \rightsquigarrow_b (e', \mathcal{H}, [e_1, \cdots, e_k])}{(\langle n;\, e\rangle, \sigma) \rightsquigarrow (\langle n;\, e'\rangle, \sigma, [\langle n;\, e_1\rangle, \cdots, \langle n;\, e_k\rangle])}$$

$$\textsc{Local-Step}\quad \frac{(e, \mathcal{H}) \rightsquigarrow_b (e', \mathcal{H}', [e_1, \cdots, e_k])}{(\langle n;\, e\rangle, (\vec{\mathcal{H}}[n \mapsto \mathcal{H}], \vec{\mathcal{S}}, \mathcal{M})) \rightsquigarrow (\langle n;\, e'\rangle, (\vec{\mathcal{H}}[n \mapsto \mathcal{H}'], \vec{\mathcal{S}}, \mathcal{M}), [\langle n;\, e_1\rangle, \cdots, \langle n;\, e_k\rangle])}$$

$$\textsc{Assign-New-Ip-Step}\quad \frac{n \neq \mathfrak{S} \qquad n \notin \mathsf{dom}(\vec{\mathcal{H}}) \qquad n \notin \mathsf{dom}(\vec{\mathcal{S}})}{(\langle \mathfrak{S};\, \texttt{start}\ n\ e\rangle, (\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M})) \rightsquigarrow (\langle \mathfrak{S};\, ()\rangle, (\vec{\mathcal{H}}[n \mapsto \emptyset], \vec{\mathcal{S}}[n \mapsto \emptyset], \mathcal{M}), [\langle n;\, e\rangle])}$$

$$\textsc{Socket-Step}\quad \frac{(e, (\mathcal{S}, \mathcal{M})) \rightsquigarrow_{(n)} (e', (\mathcal{S}', \mathcal{M}'))}{(\langle n;\, e\rangle, (\vec{\mathcal{H}}, \vec{\mathcal{S}}[n \mapsto \mathcal{S}], \mathcal{M})) \rightsquigarrow (\langle n;\, e'\rangle, (\vec{\mathcal{H}}, \vec{\mathcal{S}}[n \mapsto \mathcal{S}'], \mathcal{M}'), \epsilon)}$$

Prim-Step ($\rightarrow_p$)

Prim-Step has the signature

$$(\textit{IP-Address} \times \textit{Expr}) \times \textit{State} \rightarrow_p (\textit{IP-Address} \times \textit{Expr}) \times \textit{State} \times \textit{List}(\textit{IP-Address} \times \textit{Expr}).$$

Prim-step deals with the use of evaluation contexts as previously defined. The p in $\rightarrow_p$ is a part of the name of the relation and not a parameter.

$$\textsc{Evaluation-Context-Step}\quad \frac{(\langle n;\, e\rangle, \sigma) \rightsquigarrow (\langle n;\, e'\rangle, \sigma', [\langle n;\, e_1\rangle, \cdots, \langle n;\, e_k\rangle])}{(\langle n;\, K[e]\rangle, \sigma) \rightarrow_p (\langle n;\, K[e']\rangle, \sigma', [\langle n;\, e_1\rangle, \cdots, \langle n;\, e_k\rangle])}$$

System-Step ($\rightarrow_s$)

System-Step has the signature

$$\textit{State} \rightarrow_s \textit{State}.$$

The system-step models that the network can take a step, specifically it models that using UDP sockets, messages can be dropped, reordered and duplicated. The s in $\rightarrow_s$ is a part of the name of the relation and not a parameter.

$$\textsc{Message-Deliver}\quad \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}[n \mapsto \mathcal{S}[sh \mapsto ((\texttt{Some}\ m_{dst}, b), \mathcal{B})]], \mathcal{M}) \rightarrow_s (\vec{\mathcal{H}}, \vec{\mathcal{S}}[n \mapsto \mathcal{S}[sh \mapsto ((\texttt{Some}\ m_{dst}, b), [m] +\!\!+ \mathcal{B})]], \mathcal{M} \setminus \{m\})}$$

$$\textsc{Message-Dublicate}\ \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M}) \rightarrow_s (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{M} \uplus \{m\})} \qquad \textsc{Message-Drop}\ \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M}) \rightarrow_s (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{M} \setminus \{m\})}$$

Step ($\rightarrow$)

Step has the signature

$$\mathit{List}(\mathit{IP\text{-}Address} \times \mathit{Expr}) \times \mathit{State} \rightarrow \mathit{List}(\mathit{IP\text{-}Address} \times \mathit{Expr}) \times \mathit{State}.$$

Step is the top most relation and uses prim-step and system-step in its premises. It models that either the network will take a step or a thread on a node will take a step from the pool of all threads. T_i denotes a thread pool and σ refers to the whole state $(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{M})$.

$$\textsc{Step-Atomic}\ \frac{(\langle n;\, e\rangle, \sigma) \rightarrow_p (\langle n;\, e'\rangle, \sigma', \vec{e})}{(T_1 +\!\!+ [\langle n;\, e\rangle] +\!\!+ T_2, \sigma) \rightarrow (T_1 +\!\!+ [\langle n;\, e'\rangle] +\!\!+ T_2 +\!\!+ \vec{e}, \sigma')} \qquad \textsc{Step-State}\ \frac{\sigma \rightarrow_s \sigma'}{(T, \sigma) \rightarrow (T, \sigma')}$$

2 Program Logic

2.1 Connectives and rules

Aneris has all the types and connectives of Iris and is extended with the following:

$$\tau, \sigma ::= \ldots \mid \mathit{Handle} \mid \mathit{Socket} \mid \mathit{IP\text{-}Address} \mid \mathit{Message}$$
$$t, P, Q, \Phi ::= \ldots \mid \mathsf{FreeIp}(n) \mid \mathsf{FreePorts}(n, \mathcal{P}) \mid \ell \overset{n}{\mapsto} v \mid sh \overset{n}{\hookrightarrow} s \mid sa \rightsquigarrow (R, T) \mid sa \Mapsto \Phi \mid \mathsf{Unallocated}(A)$$

$\mathsf{FreeIp}(n)$ states that the node n (nodes are IP-addresses) is free and $\mathsf{FreePorts}(n, \mathcal{P})$ means that none of the ports in the set $\mathcal{P}$ are active on node n. $\ell \overset{n}{\mapsto} v$ and $sh \overset{n}{\hookrightarrow} s$ are respectively heap and socket mappings local to node n. $sa \rightsquigarrow (R, T)$ means that the socket address sa has received the messages in the set R and transmitted the messages in the set T. $sa \Mapsto \Phi$ denotes that the socket address sa adheres to the protocol $\Phi : \mathit{Message} \to \mathit{iProp}$ and $\mathsf{Unallocated}(A)$ states that A is a set of socket addresses not adhering to any protocol.

We present the rules of the Aneris program logic using Hoare triples denoted with the node n, on which the expression e is executing, and mask $\mathcal{E}$:

$$\{P\}\, e\, \{v.\, Q\}^n_{\mathcal{E}} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}^n_{\mathcal{E}}\, e\, \{v.\, Q\})$$

Aneris is an instantiation of Trillium (the Aneris weakest precondition is defined in terms of the Trillium weakest precondition) which is built using the Iris base logic, hence for the rules of the base logic and its derivations we refer to the Iris documentation. In addition, the usual structural rules relating weakest preconditions have been proven for Aneris (file). The significant rules of Aneris are as follows:

HT-PURE
$$\frac{(e, \sigma) \rightsquigarrow_h (e', \sigma, \epsilon) \qquad \{P\}\, e'\, \{v.\, Q\}^n_{\mathcal{E}}}{\{\triangleright P\}\, e\, \{v.\, Q\}^n_{\mathcal{E}}}$$

HT-ALLOC
$$\{\mathsf{True}\}\ \texttt{ref}\ (v)\ \{\ell.\, \ell \overset{n}{\mapsto} v\}^n_{\mathcal{E}}$$

HT-LOAD
$$\{\triangleright\, \ell \overset{n}{\mapsto} v\}\ !\ell\ \{v.\, \ell \overset{n}{\mapsto} v\}^n_{\mathcal{E}}$$

HT-STORE
$$\{\triangleright\, \ell \overset{n}{\mapsto} w\}\ \ell \leftarrow v\ \{\ell \overset{n}{\mapsto} v\}^n_{\mathcal{E}}$$

HT-FORK
$$\frac{\{P\}\, e\, \{w.\, \mathsf{True}\}^n_{\top}}{\{P\}\ \texttt{fork}\ \{e\}\ \{v.\, v = ()\}^n_{\mathcal{E}}}$$

HT-CAS-SUC
$$\{\triangleright\, \ell \overset{n}{\mapsto} v_1\}\ \texttt{CAS}\ \ell\ v_1\ v_2\ \{v.\, v = \mathsf{True} * \ell \overset{n}{\mapsto} v_2\}^n_{\mathcal{E}}$$

HT-CAS-FAIL
$$\frac{w \neq v_1}{\{\triangleright\, \ell \overset{n}{\mapsto} w\}\ \texttt{CAS}\ \ell\ v_1\ v_2\ \{v.\, v = \mathsf{False} * \ell \overset{n}{\mapsto} w\}^n_{\mathcal{E}}}$$

HT-RAND
$$\{0 < u\}\ \texttt{rand}\ u\ \{r.\, r \geq 0 \wedge r < u\}^n_{\mathcal{E}}$$

PROTOCOL-SPLIT
$$\frac{A_1 \,\#\, A_2 \qquad \mathsf{Unallocated}(A_1 \cup A_2)}{\mathsf{Unallocated}(A_1) * \mathsf{Unallocated}(A_2)}$$

PORTS-SPLIT
$$\frac{\mathcal{P}_1 \,\#\, \mathcal{P}_2 \qquad \mathsf{FreePorts}(n, \mathcal{P}_1 \cup \mathcal{P}_2)}{\mathsf{FreePorts}(n, \mathcal{P}_1) * \mathsf{FreePorts}(n, \mathcal{P}_2)}$$

HT-PROTOCOL
$$\frac{e \notin \mathit{Val} \qquad \{P * a \Mapsto \psi\}\, e\, \{v.\, Q\}^n_{\mathcal{E}}}{\{P * \mathsf{Unallocated}(\{a\})\}\, e\, \{v.\, Q\}^n_{\mathcal{E}}}$$

HT-START
$$\frac{\{P * \mathsf{FreePorts}(n, \mathcal{P})\}\, e\, \{w.\, \mathsf{True}\}^n_{\top}}{\{P * \mathsf{FreeIp}(n)\}\ \texttt{start}\ n\ e\ \{v.\, v = ()\}^{\mathfrak{S}}_{\mathcal{E}}}$$

HT-SET-BLOCKING
$$\{\triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n, p), b)\}\ \texttt{settimeout}\ sh\ 0\ 0\ \{v.\, v = () * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n, p), \texttt{true})\}^n_{\mathcal{E}}$$

HT-NEWSOCKET
$$\{\mathsf{True}\}\ \texttt{socket}\ ()\ \{sh.\, sh \overset{n}{\hookrightarrow} (\texttt{None}, \texttt{true})\}^n_{\mathcal{E}}$$

HT-SET-NON-BLOCKING

$$\frac{(0 \le x \wedge 0 \le y \wedge 0 < (x+y))}{\{\triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), b)\}}$$

$\texttt{settimeout}\ sh\ x\ y$

$\{v.\ v = () * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \texttt{false})\}^{n}_{\mathcal{E}}$

HT-SOCKETBIND

$\{\triangleright\, \mathsf{FreePorts}(n, \{p\}) * \triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{None}, b)\}$

$\texttt{socketbind}\ sh\ (n,p)$

$\{v.\ v = 0 * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), b)\}^{n}_{\mathcal{E}}$

HT-SEND

$$\left\{\begin{array}{l}(n,p) = m.\mathsf{src} * \triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ m.\mathsf{src}, b) * \\ \triangleright\, m.\mathsf{src} \rightsquigarrow (R,T) * \triangleright\, m.\mathsf{dst} \Rrightarrow \Phi\ * \triangleright \Phi(m)\end{array}\right\}$$

$\texttt{sendto}\ sh\ m.\mathsf{str}\ m.\mathsf{dst}$

$$\left\{\begin{array}{l}v.\ v = |m.\mathsf{str}| * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ m.\mathsf{src}, b) * \\ m.\mathsf{src} \rightsquigarrow (R, T \cup \{m\})\end{array}\right\}^{n}_{\mathcal{E}}$$

HT-SEND-DUPLICATE

$$\left\{\begin{array}{l}(n,p) = m.\mathsf{src} * m \in T * \triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ m.\mathsf{src}, b) * \\ \triangleright\, m.\mathsf{src} \rightsquigarrow (R,T) * \triangleright\, m.\mathsf{dst} \Rrightarrow \Phi\end{array}\right\}$$

$\texttt{sendto}\ sh\ m.\mathsf{str}\ m.\mathsf{dst}$

$$\left\{\begin{array}{l}v.\ v = |m.\mathsf{str}| * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ m.\mathsf{src}, b) * \\ m.\mathsf{src} \rightsquigarrow (R,T)\end{array}\right\}^{n}_{\mathcal{E}}$$

HT-RECV-BLOCKING

$$\left\{\begin{array}{l}\triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{True}) * \triangleright (n,p) \rightsquigarrow (R,T) * \\ (n,p) \Rrightarrow \Phi\end{array}\right\}$$

$\texttt{receivefrom}\ sh$

$$\left\{\begin{array}{l}v.\ v = \texttt{Some}\,(m.\mathsf{str}, m.\mathsf{src}) * m.\mathsf{dst} = (n,p) * \\ \Big(\big(m \notin R * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{True}) * \\ (n,p) \rightsquigarrow (R \cup \{m\}, T) * (n,p) \Rrightarrow \Phi * \Phi(m)\big) \\ \vee \big(m \in R * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{True}) * \\ (n,p) \rightsquigarrow (R,T)\big)\Big)\end{array}\right\}^{n}_{\mathcal{E}}$$

HT-RECV-NON-BLOCKING

$$\left\{\begin{array}{l}\triangleright\, sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{False}) * \triangleright (n,p) \rightsquigarrow (R,T) * \\ (n,p) \Rrightarrow \Phi\end{array}\right\}$$

$\texttt{receivefrom}\ sh$

$$\left\{\begin{array}{l}v.\ \Big(v = \texttt{None} * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{False}) * \\ \quad (n,p) \rightsquigarrow (R,T)\Big) \vee \\ \quad \Big(\exists m.\ m.\mathsf{dst} = (n,p) * v = \texttt{Some}\,(m.\mathsf{str}, m.\mathsf{src}) * \\ \quad \big((m \notin R * sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{False}) * \\ \quad (n,p) \rightsquigarrow (R \cup \{m\}, T) * \Phi(m)\big)\ \vee \big(m \in R * \\ \quad sh \overset{n}{\hookrightarrow} (\texttt{Some}\ (n,p), \mathsf{False}) * (n,p) \rightsquigarrow (R,T)\big)\Big)\end{array}\right\}^{n}_{\mathcal{E}}$$

2.2 Adequacy

Theorem (Adequacy): *Let IPs be a set of ip-addresses not containing the system node $\mathfrak{S}$, A a set of socket addresses $A \subseteq (\mathit{IP\text{-}Address} \times \mathit{Port})$, e an expression, Φ a first-order predicate on values, $\sigma = ([\mathfrak{S} \mapsto \emptyset], [\mathfrak{S} \mapsto \emptyset], \emptyset)$ the initial state of valid program execution $([\langle \mathfrak{S};\, e\rangle], \sigma) \to^{*} ([\langle \mathfrak{S};\, e'_1\rangle ... \langle n'_m;\, e'_m\rangle], \sigma')$. If*

$$\{\mathsf{Unallocated}(A) * \mathop{\Large *}_{a \in A} a \rightsquigarrow (\emptyset, \emptyset) * \mathop{\Large *}_{n \in IPs} \mathsf{FreeIp}(n))\}\, e\, \{\Phi(v)\}^{\mathfrak{S}}_{\top}$$

, then:

1. *For all i, $1 \le i \le m$, $e'_i \in \mathit{Val}$ or $\mathit{Reducible}(\langle n'_i;\, e'_i\rangle, \sigma')$*
2. *If the main thread e'_1 is a value v then $\Phi(v)$ holds at the level of the meta-logic.*