The MeTTa calculus

L.G. Meredith[1]

CEO, F1R3FLY.io 9336 California Ave SW, Seattle, WA 98103, USA,
ceo@f1r3fly.io

Abstract. We describe a core calculus that captures the operational semantics of the language MeTTa. We show how to render this operational semantics to efficient implementation on modern computers. We derive a security model, transaction model, spatial-behavioral type system, and stochastic and quantum execution paradigms from the calculus.

1 Introduction and motivation

In [?] we described a register machine based operational semantics for MeTTa. While this has some utility for proving the correctness of compilers, it is not conducive for reasoning about types and other more abstract aspects of MeTTa-based computation. Here we present a calculus, together with an efficient implementation and prove the correctness of the implementation.

Additionally, we use the OSLF algorithm developed by Meredith, Stay, and Williams to calculate a spatial-behavioral type system for MeTTa. Further, we adapt a well known procedure for proving the termination of rewrites to provide a token-based security model for the calculus and implementation. Beyond that we derive versions of fuzzy, stochastic, and quantum execution modes, automatically.

In general, presenting MeTTa as a graph structured lambda theory, otherwise known as a structured operational semantics, not only has the benefit that implementation follows the correct-by-construction methodology, but may be used to automatically derive and extend MeTTa with much needed features for programming real applications in a distributed and decentralized setting, as well as supporting well established programming paradigms, such as semi-colon delimited sequential assignment programs.

2 A symmetric reflective higher order concurrent calculus with backchaining

Note, in the following spec we use $[e]$ to denote a space-delimited finite sequence of e's; and we use $[e]_{seq}$ to denote a comma-delimited finite sequence of e's.

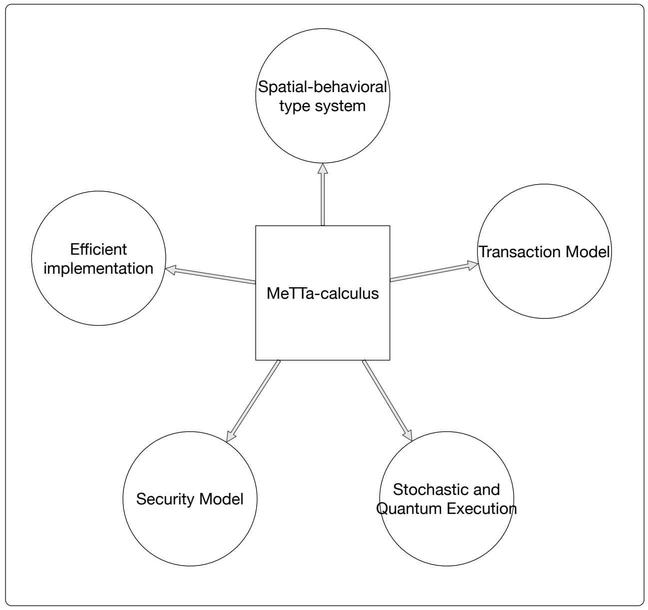

Fig. 1. Generating features from the MeTTa-calculus

PROCESS
$$P, Q ::= 0 \mid G \mid \mathsf{for}(t \texttt{<->} x)P \mid x?P \mid *x \mid P|Q$$

NAME
$$x, y ::= @P$$

TERM
$$t, u ::= atom \mid ([t])$$

ATOM
$$atom ::= x \mid P$$

GROUND
$$G ::= \mathsf{BoolLiteral} \mid \mathsf{StringLiteral} \mid \mathsf{IntLiteral} \mid C$$

COLLECTION
$$C ::= \texttt{[}[t]_{seq}\texttt{]} \mid \texttt{(}t\ \texttt{,}[t]_{seq}\texttt{)} \mid \texttt{Set(}[t]_{seq}\texttt{)} \mid \{[t : t]\}$$

EQUIV
$$P|0 \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)\mathsf{R}$$

ALPHA
$$\frac{\mathsf{occurs}(t, y)}{\mathsf{for}(t \texttt{<->} x)P \equiv \mathsf{for}(t\{z/y\} \texttt{<->} x)(P\{z/y\}) \ \mathsf{if}\, z \notin \mathsf{FN}(P)}$$

$$\text{COMM}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\mathsf{for}(t\texttt{<->}x)P \mid \mathsf{for}(u\texttt{<->}x)Q \to P\dot{\sigma}|Q\dot{\sigma}}$$

$$\text{PAR}\quad \frac{P \to P'}{P|Q \to P'|Q} \qquad \text{EQUIV}\quad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

$$\text{REFL}\quad \frac{P \to P'}{x?P \to \mathsf{for}((P')\texttt{<->}x)0}$$

where $\dot{\sigma}$ denotes the substitution that replaces all variable to process bindings with variable to name bindings. Thus, $\dot{\{P/x\}} = \{@P/x\}$.

We denote the collection of process states generated (resp. recognized) by this grammar by Proc. Likewise, we denote the collection of spaces (aka channels or names) by @Proc.

2.1 Intuitive mapping to MeTTa

MeTTa language	MeTTa calculus
(addAtom *space term*)	$\mathsf{for}(term \texttt{<->} space)0$
(remAtom *space term*)	$\mathsf{for}(term \texttt{<->} space)0$
(? *space term*)	$\mathsf{for}(term \texttt{<->} space)0$

The key insight is the same one that Google and other Web2.0 companies made decades ago: $folders = tagging$. In other words, rather than actually building a container data structure constituting a "space" (as in AtomSpace), we merely tag atoms with the space(s) they occupy. This shift in perspective allows us to use the same construct (a kind of tagging) for adding atoms to a space; removing atoms from a space; and, querying for atoms in a space that match a given pattern.

Likewise a rewrite rule is a continuation $t\texttt{<->}\{\ P\ \}$. When it is tagged with a space, say x, i.e. $\mathsf{for}(t\texttt{<->}x)P$, may be thought of as added to the space.

Under this view, a space is equated with all the atoms (and rules) that have been tagged as in the space. That is, we may define a function $\mathsf{space} : @\mathsf{Proc} \to \mathsf{Proc}$ by $\mathsf{space}(x) = \Pi_i\ \mathsf{for}(t_i\texttt{<->}x)P_i$. Once we make this definition we note we have two interlated algebras of considerable additional interest. One is the algebra of spaces as tags, which is isomorphic to the algebra of process states, and another is the algebra of the spaces as collections of atoms (and rules).

The first affords the ability to programmatically define tags and filter on tags. The second affords the ability to reason about, filter, and traverse spaces-qua-collections. The fact that these two are interrelated means that spaces can be nested, or more generally composed in a variety of interesting ways. Indeed, some collections of spaces may be mutually recursively defined!

3 Some useful features

3.1 Replication and freshness

$$\begin{array}{l}\text{PROCESS}\\ P,Q ::= \ldots \mid\ !P \mid \mathsf{new}\ x\ \mathsf{in}\ \{\ P\ \}\end{array}$$

In the core calculus, when two terms rendezvous at a space ($\mathsf{for}(t \text{<->} x)P \mid \mathsf{for}(u \text{<->} x)Q$) they are *consumed* and replaced by their continuations ($P\dot{\sigma}|Q\dot{\sigma}$). It is frequently useful in programming applications to leave one or the other in place. Thus, when $!\mathsf{for}(t \text{<->} x)P$ rendezvous with $\mathsf{for}(u \text{<->} x)Q$ it reduces to $!\mathsf{for}(t \text{<->} x)P|P\dot{\sigma}|Q\dot{\sigma}$.

Likewise, in programming applications it is often useful to guarantee that computations rendezvous in a private space. The state denoted by $\mathsf{new}\ x\ \mathsf{in}\ \{\ P\ \}$ guarantees that x is private in the scope P. Therefore, $\mathsf{new}\ x\ \mathsf{in}\ \{\ \mathsf{for}(t \text{<->} x)P \mid \mathsf{for}(u \text{<->} x)Q\}$ guarantees that the rendezvous happens in a private space.

3.2 Fork-join concurrency

This next bit of syntactic sugar illustrates the value of the for-comprehension. Specifically, it facilitates the introduction of fork-join concurrency, which is predominant in human decision-making processes. The following syntax should be read as an expansion of the core calculus, *replacing* the much simpler for-comprehension with a more articulated one.

$$\begin{array}{l}\text{PROCESS}\\ P,Q ::= \ldots \mid \mathsf{for}([\mathsf{Join}])P\end{array}$$

$$\begin{array}{l}\text{JOINS}\\ [\mathsf{Join}] ::= \mathsf{Join} \mid \mathsf{Join};[\mathsf{Join}]\end{array}$$

$$\begin{array}{l}\text{JOIN}\\ \mathsf{Join} ::= [\mathsf{Query}]\end{array}$$

$$\begin{array}{l}\text{QUERIES}\\ [\mathsf{Query}] ::= \mathsf{Query} \mid \mathsf{Query}\&[\mathsf{Query}]\end{array}$$

$$\begin{array}{l}\text{QUERY}\\ \mathsf{Query} ::= t \text{ <-> } x\end{array}$$

In case the BNF is a little opaque, here is the template.

```
for(
  y11 <-> x11 & ... & ym1 <-> xm1 ; // received in any order,
     ... ; // but all received before the next row
  y1n <-> x1n & ... & ymn <-> xmn
){ P }
```

As mentioned previously, the predominant pattern of human decision making processes (such as loan approval processes, or academic paper reviews) involve fork-join concurrency. The syntactic sugar provided here certainly supports that kind of coordination amongst processes. For example,

```
for (
    // received in any order
    true <-> reviewer_1 & true <-> reviewer_2 & true <-> reviewer_3
    ){
      // acceptance notification and publication process
      P
    }
```

Yet, it affords much more sophisticated control than this, while also providing a programming paradigm that is familiar to modern programmers, namely semi-colon delimited sequential assignment-based programming.

3.3 The COMM rule as transactional semantics

$$\text{COMM}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\mathsf{for}(t \mathtt{<->} x)P \mid \mathsf{for}(u \mathtt{<->} x)Q \rightarrow P\dot{\sigma} | Q\dot{\sigma}}$$

One of the most interesting aspects of these types of operational semantics [1] is that they are implicitly transactional. In the asymmetric case it is easier to see because one of the interacting items is a write to a space and the other is a read. In the symmetric case both threads are reading and writing at the same time. Specifically, all the variables in t that are substituted for by terms in u are being read from u and written to the continuation by the substitution $P\dot{\sigma}$; and symmetrically, all the variables in u that are substituted for by terms in t are being read from t and written to the continuation $Q\dot{\sigma}$. Indeed, $\dot{\sigma}$ is the witness of the transaction.

4 Compiling MeTTa code to the MeTTa-calculus

TBD

5 From calculus to efficient implementation

It turns out that a variant of the Linda tuple space [?] where input is not blocking is the critical innovation to an efficient implementation. Instead of blocking we turn input from a key into the storage of a continuation at the key. As the astute reader has already guessed, (the hashes of) channels become keys.

[1] such as are seen in the π-calculus or the rho calculus

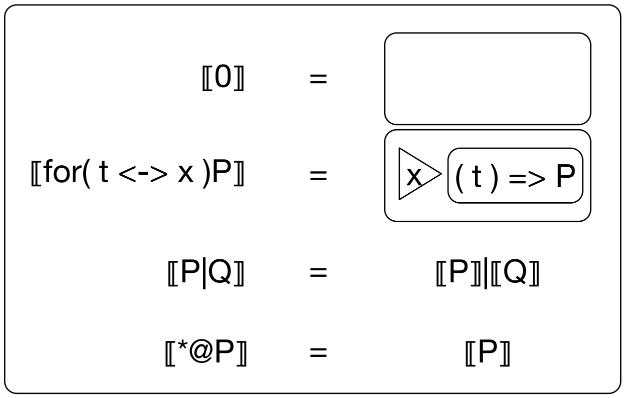

Fig. 2. Compiler from MeTTa-calculus to RSpace

Thus, the diagram above indicates how to turn a given process state into an RSpace instance.

- the first equation says that the 0 process corresponds to the empty RSpace;
- the second says that a comprehension corresponds to an RSpace consisting of a single key-value pair; the key being the hash of the channel on which the comprehension is listening (the *source* of the comprehension); and the value being the (multiset containing the single) continuation formed by created a pattern-matching style lambda from *target* of the comprehension to its *body*; [2]
- the third equation translates the concurrent composition of two processes, say P and Q, into the combination of their two RSpaces using an operation RSpaces on we define below.

Parallel composition of RSpaces If we are combining two RSpaces that only have a single key-value pair, each; and the keys are not equal, then we simply combing them into a single RSpace containing both key-value pairs. More generally, when combining RSpaces that have no overlap in their key-sets, we simply return the RSpace whose key-value pairs is the union of the key-value pairs of the RSpaces being combined.

When combining two RSpaces that only have a single key-value pair, each, their keys are the same, but the patterns of their continuations do not unify, then we simply combine them into a single RSpace containing a single key-value pair, the key of which is the key of each pair (remember, both pairs share a key) and the value of which is the multiset containing the respective values (continuations).

More generally, when combining RSpaces that have overlap in their key-sets, but the overlapping keys contain no continuations whose patterns unify, we simply return the RSpace whose key-value pairs is

[2] Sometimes we abuse the terminology and call the body the continuation.

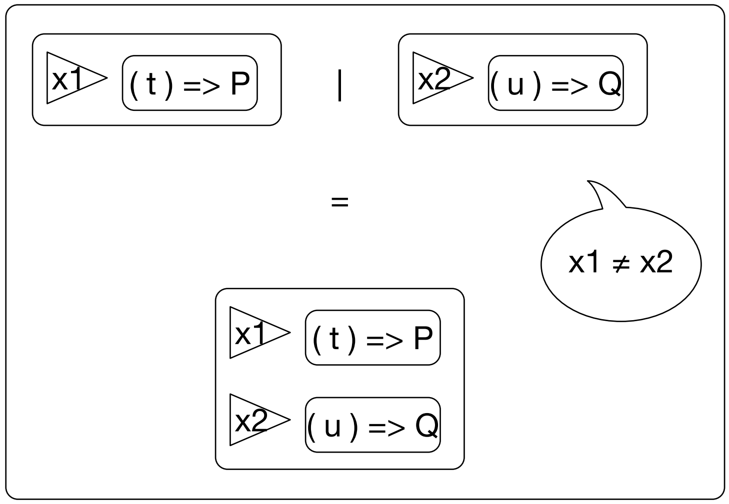

Fig. 3. Compiler from MeTTa-calculus to RSpace continued

- the union of the key-value pairs of the RSpaces being combined for the key-value pairs whose keys do not overlap;
- and constructs a new key-value pair for each pair in the overlap that shares a key, whose key is the key they share and whose value is the union of their respective multisets.

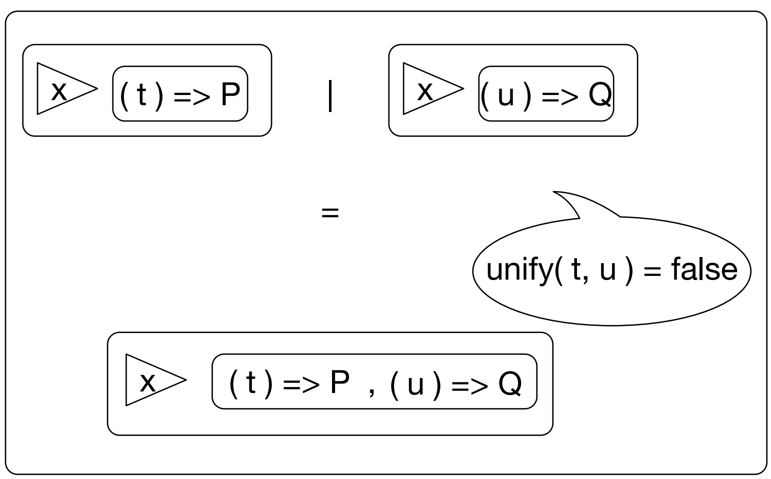

Fig. 4. Compiler from MeTTa-calculus to RSpace continued again

What remains is the case when combining two key-value pairs that have unifying patterns in their multisets. The naive algorithm is quadratic in the number of unification checks. However, MORK provides a more efficient way to parallelize these checks.

[MORK algorithm description goes here.]

Procedural reflection implementation The astute reader will have noticed that we have not addressed the execution of procedurally reflective processes. The naive implementation instantiates a *copy* of the RSpace in which $x?P$ is being evaluated and allows one transactonal step in the future of P, i.e. one COMM event in the future of P. Then it makes that state available at x in the original RSpace.

Unfortunately, there is no way to avoid the cost of copying the original RSpace. We should note that there is a natual way to shift all processes into a different namespace.

$$u * P = P\{u * x/x | x \in \mathsf{FN}(P)\}$$

$$@P * @Q = @(P|Q)$$

Thus, all the key-value pairs could remain in the original RSpace and merely be shifted into a distinguished namespace. However, this would actually be more costly than a deep copy of the original RSpace because the key-value pairs still have to be copied and then shifted.

5.1 The transactional semantics of RSpace

A given RSpace represents a collection of MeTTa spaces that share a transactional semantics. Effectively, the collection of channels served by the RSpace, that is the *namespace* served by a given RSpace, all participate in coordinated transactions in the same way that all the tables served by a SQL server participate in coordinated transactions. However, unlike SQL's architecture, the RSpace architecture naturally composes, allowing for a hierarchy of transactional coordination over a tree of namespaces served by a network of RSpaces.

6 Tokenized security

This tokenized security model is *generated* from the operational semantics.

SECURITY-TOKENS
$$T ::= () \mid s \mid T : T$$

SECURED-PROCESSES
$$S ::= \{P\}_s \mid T \mid S|S$$

MULTI-PARTY-SIGS
$$s ::= () \mid \mathsf{hash}(<signature>) \mid s\&s$$

where $\{P\}_s$ is a process signed by a digital signature.

COMM-COSIGNED-PAR-EXTERNAL-SEQUENTIAL

$$\frac{\sigma = \mathsf{unify}(t,u)}{\{\mathsf{for}(t \texttt{<->} x)P\}_{s_1} \mid \{\mathsf{for}(u \texttt{<->} x)P\}_{s_2} \mid s_1 \& s_2 : T \rightarrow \{P\dot{\sigma}|Q\dot{\sigma}\}_{s_1 \& s_2} \mid T}$$

COMM-COSIGNED-PAR-EXTERNAL-CONCURRENT

$$\frac{\sigma = \mathsf{unify}(t,u)}{\{\mathsf{for}(t \texttt{<->} x)P\}_{s_1} \mid \{\mathsf{for}(u \texttt{<->} x)P\}_{s_2} \mid s_1 : T_1 \mid s_2 : T_1 \rightarrow \{P\dot{\sigma}|Q\dot{\sigma}\}_{s_1 \& s_2} \mid T_1 \mid T_2}$$

COMM-SIGNED

$$\frac{P \rightarrow P'}{\{P\}_s \mid s : T \rightarrow \{P'\}_s \mid T}$$

COMM-COSIGNED-PAR-INTERNAL

$$\frac{P \rightarrow P'}{\{P\}_{s_1 \& s_2} \mid s_1 : T_1 \mid s_2 : T_2 \rightarrow \{P'\}_s \mid T_1 \mid T_2}$$

7 Spatial-behavioral types

This type system is *generated* from the operational semantics by Meredith and Stay's OSLF algorithm.

7.1 Type syntax

PROCESS-TYPE

$$T, U ::= \mathsf{0} \mid GT \mid \langle (TT \texttt{<->} N) \rangle T \mid \langle x? \rangle T \mid *N \mid T|U$$

NAME-TYPE

$$N ::= @T$$

TERM-TYPE

$$TT ::= AtomT \mid ([T])$$

ATOM-TYPE

$$AtomT ::= N \mid T$$

GROUND-TYPE

$$GT ::= \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Int} \mid C$$

COLLECTION-TYPE

$$C ::= \mathsf{List}(TT) \mid \mathsf{Tuple}(TT\ ,[TT]_{seq}) \mid \mathsf{Set}(TT) \mid \mathsf{Map}\{[TT : TT]\}$$

7.2 Type judgments and type inferences

We adopt a standard syntax for judgments, but use a horizontal syntax for inference to conserve vertical space or the more standard vertical syntax to conserve horizontal space. The two formats are just different ways of writing a type inference rule that depends on a collection of judgments to establish a judgment.

Note that the rules below are the ones specific to the MeTTa-calculus. The OSLF algorithm generates a host of ambient typing judgments that are common to all such systems by the fact that they are expressible as graph structured lambda theories.

JUDGMENT

$$Jugment ::= \mathit{\Gamma} \vdash P : T$$

INFERENCE-HORIZONTAL

$$Inference ::= [Judgment] \Vdash Judgment$$

INFERENCE-VERTICAL

$$\frac{[Judgment]}{Judgment}$$

GROUND-BOOL

$$\Vdash \mathsf{Boolliteral} : \mathsf{Bool}$$

GROUND-STRING

$$\Vdash \mathsf{Stringliteral} : \mathsf{String}$$

GROUND-INT

$$\Vdash \mathsf{Intliteral} : \mathsf{Int}$$

COLLECTION-LIST

$$\mathit{\Gamma}_1 \vdash t_1 : TT \ldots \mathit{\Gamma}_n \vdash t_n : TT \Vdash \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n \vdash [t_1, \ldots, t_n] : \mathsf{List}[TT]$$

COLLECTION-TUPLE

$$\mathit{\Gamma}_1 \vdash t_1 : TT_1 \ldots \mathit{\Gamma}_n \vdash t_n : TT_n \Vdash \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n \vdash (t_1, \ldots, t_n) : \mathsf{Tuple}(TT_1, \ldots, TT_n)$$

COLLECTION-SET

$$\mathit{\Gamma}_1 \vdash t_1 : TT \ldots \mathit{\Gamma}_n \vdash t_n : TT \Vdash \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n \vdash \mathsf{Set}(t_1, \ldots, t_n) : \mathsf{Set}(TT)$$

COLLECTION-MAP

$$\frac{\mathit{\Gamma}_1 \vdash k_1 : TT_1 \mathit{\Delta}_1 \vdash v_1 : TT_2 \ldots \mathit{\Gamma}_n \vdash k_n : TT_{2n-1} \mathit{\Delta}_n \vdash v_n : TT_{2n}}{\mathit{\Gamma}_1, \mathit{\Delta}_n \ldots, \mathit{\Gamma}_n, \mathit{\Delta}_n \vdash \mathsf{Map}\{k_1 : v_1 \ldots k_n : v_n\} : \mathsf{Map}\{TT_1 : TT_2 \ \ldots \ TT_{2n-1} : TT_{2n}\}}$$

FOR-COMPREHENSION

$$t : TT, \mathit{\Gamma} \vdash P : T \quad \mathit{\Delta} \vdash x : V \Vdash \mathit{\Gamma}, \mathit{\Delta} \vdash \mathsf{for}(t \texttt{<->} x)P : \langle (TT \texttt{<->} V) \rangle T$$

REFL

$$\mathit{\Gamma} \vdash P : T \quad \mathit{\Delta} \vdash x : V \Vdash \mathit{\Gamma}, \mathit{\Delta} \vdash x?P : \langle ?V \rangle T$$

PARALLEL

$$\mathit{\Gamma} \vdash P : T \quad \mathit{\Delta} \vdash Q : U \Vdash \mathit{\Gamma}, \mathit{\Delta} \vdash P|Q : T|U$$

DEREF

$$\mathit{\Gamma} \vdash x : V \Vdash \mathit{\Gamma} \vdash *x : *V$$

QUOTE

$$\mathit{\Gamma} \vdash P : T \Vdash \mathit{\Gamma} \vdash @x : @T$$

Equations

ParMonoidId

$$\frac{\Gamma \vdash P : T}{\Gamma \vdash P|0 = P : T}$$

ParMonoidAssoc

$$\frac{\Gamma \vdash (P|Q)|R : T}{\Gamma \vdash (P|Q)|R = P|(Q|R) : T}$$

ParMonoidComm

$$\frac{\Gamma \vdash P|Q : T}{\Gamma \vdash P|Q = Q|P : T}$$

Lifted redex constuctors

Comm

$$\frac{\Gamma_1 \vdash x : V \quad t_1 : TT_1, \Gamma_2 \vdash P : T_1 \quad t_2 : TT_2, \Gamma_3 \vdash Q : T_2}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{comm}(x, t, u, P, Q) : \mathsf{comm}(V, TT_1, TT_2, T_1, T_2)}$$

Eval

$$\Gamma \vdash P : T \Vdash \Gamma \vdash \mathsf{eval}(P) : \mathsf{eval}(T)$$

ParL

$$\Gamma \vdash R : E \quad \Delta \vdash P : T \Vdash \Gamma, \Delta \vdash \mathsf{par}_L(R, P) : \mathsf{par}_L(E, T)$$

ParR

$$\Gamma \vdash R : E \quad \Delta \vdash P : T : \Vdash \Gamma, \Delta \vdash \mathsf{par}_R(P, R) : \mathsf{par}_R(T, E)$$

Reductions from the $\boldsymbol{\lambda}$-theory

COMM-SRC

$$\frac{\Gamma \vdash \mathsf{comm}(x,t,u,P,Q) : \mathsf{comm}(V,TT_1,TT_2,T,U)}{\Gamma \vdash \mathsf{src}(\mathsf{comm}(x,t,u,P,Q)) = \mathsf{for}(t \Leftrightarrow x)P|\mathsf{for}(u \Leftrightarrow x)Q : \langle TT_1 \Leftrightarrow V\rangle T|\langle TT_2 \Leftrightarrow V\rangle U}$$

COMM-TRGT

$$\frac{\Gamma \vdash \mathsf{comm}(x,t,u,P,Q) : \mathsf{comm}(V,TT_1,TT_2,T,U)}{\Gamma \vdash \mathsf{trgt}(\mathsf{comm}(x,t,u,P,Q)) = P\dot{\sigma}|Q\dot{\sigma} : T|U}$$

EVAL-SRC

$$\Gamma \vdash \mathsf{eval}(P) : \mathsf{eval}(T) \Vdash \Gamma \vdash \mathsf{src}(\mathsf{eval}(P)) = *@P : *@\mathbf{T}$$

EVAL-TRGT

$$\Gamma \vdash \mathsf{eval}(P) : \mathbf{eval}(\mathbf{U}) : \mathbf{R} \Vdash \Gamma \vdash \mathsf{trgt}(\mathsf{eval}(P)) = P : U$$

PAR-SRC

$$\frac{\Gamma \vdash \mathsf{par}(R,P) : \mathbf{par}(E,T)}{\Gamma \vdash \mathsf{src}(\mathsf{par}(R,P)) = \mathsf{src}(R)|P : src(E)|T}$$

PAR-TRGT

$$\frac{\Gamma \vdash \mathsf{par}(R,P) : \mathbf{par}(E,T)}{\Gamma \vdash \mathsf{trgt}(\mathsf{par}(R,P)) = \mathsf{trgt}(R)|P : trgt(E)|T}$$

8 Stochastic and quantum execution

This following stochastic and quantum execution mechanisms are *generated* from the operational semantics by Meredith, Stay, and Warrell's algorithm.

8.1 Stochastic execution

TBD

8.2 Quantum execution

TBD

9 Conclusions and future research

TBD

Acknowledgments. The author wishes to thank SingularityNet.io for their support of this work.