Meta-MeTTa: an operational semantics for MeTTa

Lucius Gregory Meredith[1]
, Ben Goertzel[2], and Jonathan Warrell[3]

[1] CEO, F1R3FLY.io 9336 California Ave SW, Seattle, WA 98103, USA
f1r3fly.ceo.com
[2] CEO, SingularityNet
ben@singularitynet.io
[3] Researcher, SingularityNet
jonathan.warrell@singularitynet.io

Abstract. We present an operational semantics for the language MeTTa and use it to prove the correctness of a translation of MeTTa into rholang.

1 Introduction and motivation

We present joint work between SingularityNet.io and F1R3FLY.io. As part of their OpenCog and Hyperon projects SingularityNet.io [3] [2] has developed the language MeTTa in which humans and AGIs codify the behavior of AGIs. F1R3FLY.io is providing a compilation from MeTTa to rholang, a highly concurrent language, based on the rho-calculus, designed for high throughput and scalable transaction servers [7] [6].

To facilitate a correct compilation, both teams have developed a formal operational semantics for MeTTa. Here we summarize the semantics and the methodology used to develop it and demonstrate how it facilitates a provably correct compilation into rholang.

2 Examples from computer science

Since Milner's seminal Functions as processes paper, the gold standard for a presentation of an operational semantics is to present the algebra of states via a grammar (a monad) and a structural congruence (an algebra of the monad), and the rewrite rules in Plotkin-style SOS format [8] [9].

λ-calculus

Algebra of States

PROGRAM
$$Term[V] ::= V \ \mid \ \lambda V.Term[V] \ \mid \ (Term[V]\ Term[V])$$

The structural congruence is the usual α-equivalence, namely that $\lambda x.M \equiv \lambda y.(M\{y/x\})$ when y not free in M.

It is evident that $Term[V]$ is a monad and imposing α-equivalence gives an algebra of the monad.

Transitions The rewrite rule is the well know β-reduction.

$$\frac{}{((\lambda x.M)N) \to M\{N/x\}}\ \textsc{Beta}$$

π-calculus

Algebra of States

$$\textsc{Process}\quad Term[N] ::= \mathsf{0} \mid \mathsf{for}(N \leftarrow N)Term[N] \mid N!(N) \mid (\mathsf{new}\ N)Term[N] \mid Term[N]|Term[N] \mid\ !Term[N]$$

The structural congruence is the smallest equivalence relation including α-equivalence making $(Term[N], \mid, \mathsf{0})$ a commutative monoid, and respecting

$$(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}\ x)P$$
$$(\mathsf{new}\ x)(\mathsf{new}\ y)P \equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P$$
$$((\mathsf{new}\ x)P)|Q \equiv (\mathsf{new}\ x)(P|Q), x \notin \mathsf{FN}(Q)$$

Again, it is evident that $Term[N]$ is a monad and imposing the structural congruence gives an algebra of the monad.

Transitions The rewrite rules divide into a core rule, and when rewrites apply in context.

$$\frac{}{\mathsf{for}(y \leftarrow x)P|x!(z) \to P\{z/y\}}\ \textsc{comm}$$

$$\frac{P \to P'}{(\mathsf{new}\ x)P \to (\mathsf{new}\ x)P'}\ \textsc{par} \qquad \frac{P \to P'}{P|Q \to P'|Q}\ \textsc{par}$$

$$\frac{P \equiv P' \to Q' \equiv Q}{P \to Q}\ \textsc{struct}$$

For details see [8]

rho-calculus

Algebra of States Note that the rho-calculus is different from the λ-calculus and the π-calculus because it is *not* dependent on a type of variables or names. However, it does give us the opportunity to expose how ground types, such as Booleans, numeric and string operations are imported into the calculus. The calculus does depend on the notion of a 0 process. In fact, this could be any builtin functionality. The language rholang, derived

from the rho-calculus, imports all literals as *processes*. Note that this is in the spirit of the λ-calculus and languages derived from it: Booleans, numbers, and strings are terms, on the level with λ terms.

So, the parameter to the monad for the rho-calculus takes the collection of builtin processes. Naturally, for all builtin processes other than 0 there have to be reduction rules. For brevity, we take $Z = \{0\}$.

PROCESS
$$Term[Z] ::= Z \mid \mathsf{for}(Name[Z] \leftarrow Name[Z])Term[Z] \mid Name[Z]!(Term[Z]) \mid *Name[Z] \mid Term[Z]|Term[Z]$$

NAME
$$Name[Z] ::= @Term[Z]$$

The structural congruence is the smallest equivalence relation including α-equivalence making $(P, \mid, \mathsf{0})$ a commutative monoid.

Again, it is evident that $Term[Z]$ is a monad and imposing the structural congruence gives an algebra of the monad.

Transitions The rewrite rules divide into a core rule, and when rewrites apply in context.

COMM
$$\frac{x_t \equiv_{\mathsf{N}} x_s}{\mathsf{for}(y \leftarrow x_t)P \mid x_s!(Q) \rightarrow P\{@Q/y\}}$$

PAR
$$\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

STRUCT
$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

For details see [6].

Register machines and WYSIWYG semantics One important point about the previous three examples is that they are all examples of WYSIWYG operational semantics in the sense that the states *are* the terms of the calculi. Register machines, such as the famous SECD machine, or more industrially robust machines, like the JVM, are not strictly WYSIWYG. [1] [5] In fact, the register based states are not strictly a monad, but contain a monad. In particular, the terms in the language are only part of the state, which includes elements such as a stack, a heap, or other registers.

WYSIWYG models make static analysis dramatically simpler. Specifically, an analyzer only has to look at terms in the language. Language designs that do not offer a WYSIWYG presentation typically cannot be analyzed entirely through the terms of the language. In those cases properties like safety, liveness, or security are properties of the states, not just the program terms.

3 A presentation of the semantics of MeTTa

Accordingly, we present the semantics of MeTTa in terms of a monad describing the algebra of states, a structural equivalence quotienting the algebra of states, and some rewrite rules describing state transitions.

3.1 Rationale for such a presentation

The rationale for such a presentation is not simply that this is the way it's done. Instead, the benefits include

- an effective (if undecidable) notion of program equality;
- an independent specification allowing independent implementations;
- meta-level computation, including type checking, model checking, macros, computational reflection, etc.

Effective program equality Of course, the notion we are calling program equality is called bisimulation in the literature. One of the key benefits of having a notion of bisimulation explicitly spelled out is that it makes possible both by-hand and automated proofs of correctness of implementations of MeTTa. We illustrate this in a later section of the paper where we specify a compilation from MeTTa to the rho-calculus and rholang and provide both a clear statement of what it means for the compiler to be correct and a proof that it is so.

Independent implementations Hand in hand with being able to prove an implementation correct is the ability to support multiple independent implementations, each of which is provably in compliance with the specification. Pioneered by efforts like the JVM, this approach has been remarkably effective in modern projects, like Ethereum, where the Yellow Paper made it possible for many independent teams to implement compilant Ethereum clients. [11] This network of clients is regularly responsible for the deployment and correct execution of millions of $ of transactions each month.

Perhaps more salient to the MeTTa developer community, it means that no one project needs to do all the development of MeTTa clients. This spreads not only the cost of development around, but the risk. In a word, it makes MeTTa more robust against the failure of any one project or team. Correct-by-construction methodology and the tools of operational semantics dramatically enhance the already proven power of open source development to decentralize cost and risk.

3.2 Meta-level computation

Presumably efforts by human developers to develop provably correct compilation schemes from MeTTa to other computational models are just the first generation of intelligences that will seek to do so. Over time the hope is that many different kinds of intelligences will model, and hence make amenable to adaptation, MeTTa's model of computation. In particular, AGI's will seek to do the same and at dramatically different scales and timeframes.

3.3 MeTTa Operational Semantics

The complexity of MeTTa's operational semantics is somewhere between the simplicity of the λ-calculus and the enormity of the JVM. Note that MeTTa is not WYSIWYG, however, it is not such a stretch to make a version of MeTTa that is.

Algebra of States

Terms

AtomSpace

$$Term ::= (Term\ [Term]) \mid \{Term\ [Term]\} \mid Atom \mid () \mid \{\}$$

We impose the equation $\{\ldots, t, u, \ldots\} = \{\ldots, u, t, \ldots\}$, making terms of this form multisets. Note that for multiset comprehensions this amounts to non-determinism in the order of the terms delivered, but they are still streams. We use $\{Term\}$ to denote the set of terms that are (extensionally or intensionally) defined multisets, and $(Term)$ to denote the set of terms that are (extensionally or intensionally) defined lists. Informally, we will refer to elements in $\{Term\}$ as spaces.

We assume a number of polymorphic operators, such as ++ which acts as union on multisets and append on lists and concatenation on strings, and :: which acts as cons on lists and the appropriate generalization for the other data types.

Extensionally vs intensionally defined spaces The complete operational semantics makes a distinction between extensionally defined spaces and terms, where each element of the space or term has been explicitly constructed, versus intensionally defined spaces and terms where elements are defined by a rule. The latter we call comprehensions.

This design is adopted for numerous reasons:

- it provides an explicit representation for bindings;
- it provides an explicit representation for infinite terms and spaces;
- it provides an explicit scope for access to remotely accessed data.

The reader is encouraged to read the full semantics for more details. In this summary we elide the presentation of intensionally defined spaces and terms.

Term sequences

Sequence

$$[Term] ::= \epsilon \mid Term \mid Term\ [Term]$$

Literals and builtins

Atom

$$Atom ::= Ground \mid Builtin \mid Var$$

Boolean

$$BoolLiteral ::= \mathsf{true} \mid \mathsf{false}$$

Ground

$$Ground ::= BoolLiteral \mid LongLiteral \mid StringLiteral \mid UriLiteral$$

Builtin

$$Builtin ::= \mid = \mid \mathsf{transform} \mid \mathsf{addAtom} \mid \mathsf{remAtom}$$

Var

$$TermVar ::= \_ \mid Var$$

States

$State ::= \langle \{Term\}, \{Term\}, \{Term\}, \{Term\} \rangle$

We will use S, T, U to range over states and $\mathsf{i} := \pi_1$, $\mathsf{k} := \pi_2$, $\mathsf{w} := \pi_3$, and $\mathsf{o} := \pi_4$ for the first, second, third, and fourth projections as accessors for the components of states. Substitutions are ranged over by σ, and as is standard, substitution application will be written postfix, e.g. $t\sigma$.

A state should be thought of as consisting of 4 *registers*: i is the input register where queries are issued; k is the knowledge base; w is a workspace; o is the output register. We separate the input, workspace, and output registers to allow for coarse-graining of bisimulation. An external agent cannot necessarily observe the transitions related to the workspace.

Rewrite Rules

QUERY

$$\frac{\sigma_i = \mathsf{unify}(t', t_i), k = \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k', \mathsf{insensitive}(t', k')}{\langle \{t'\} \mathbin{+\!\!+} i, k, w, o \rangle \to \langle i, k, \{u_1\sigma_1\} \mathbin{+\!\!+} \ \ldots\ \mathbin{+\!\!+} \{u_n\sigma_n\} \mathbin{+\!\!+} w, o \rangle}$$

CHAIN

$$\frac{\sigma_i = \mathsf{unify}(u, t_i), k = \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k', \mathsf{insensitive}(u, k')}{\langle i, k, \{u\} \mathbin{+\!\!+} w, o \rangle \to \langle i, k, \{u_1\sigma_1\} \mathbin{+\!\!+} \ \ldots\ \mathbin{+\!\!+} \{u_n\sigma_n\} \mathbin{+\!\!+} w, o \rangle}$$

TRANSFORM

$$\frac{\sigma_i = \mathsf{unify}(t, t_i), k = \{t_1, \ldots, t_n\} \mathbin{+\!\!+} k', \mathsf{insensitive}(t, k')}{\langle \{(\mathsf{transform}\ t\ u)\} \mathbin{+\!\!+} i, k, w, o \rangle \to \langle i, k, \{u\sigma_1\} \mathbin{+\!\!+} \ \ldots\ \mathbin{+\!\!+} \{u\sigma_n\} \mathbin{+\!\!+} w, o \rangle}$$

ADDATOM1

$$\langle \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} i, k, w, o \rangle \to \langle i, k \mathbin{+\!\!+} \{t\}, w, \{()\} \mathbin{+\!\!+} o \rangle$$

ADDATOM2

$$\frac{\langle i_1, k_1, w_1, o_1 \rangle \to \langle i_2, k_2, w_2, o_2 \rangle, k_3 = \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} k_1}{\langle i_1, k_3, w_1, o_1 \rangle \to \langle i_2, \{(\mathsf{addAtom}\ t), t\} \mathbin{+\!\!+} k_2, w_2, \{()\} \mathbin{+\!\!+} o_2 \rangle}$$

REMATOM1

$$\langle \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} i, \{t\} \mathbin{+\!\!+} k, w, o \rangle \to \langle i, k, w, \{()\} \mathbin{+\!\!+} o \rangle$$

REMATOM2

$$\frac{\langle i_1, k_1, w_1, o_1 \rangle \to \langle i_2, k_2, w_2, o_2 \rangle, k_3 = \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} \{t\} \mathbin{+\!\!+} k_1}{\langle i_1, k_3, w_1, o_1 \rangle \to \langle i_2, \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} k_2, w_2, \{()\} \mathbin{+\!\!+} o_2 \rangle}$$

OUTPUT

$$\frac{\mathsf{insensitive}(u, k)}{\langle i, k, \{u\} \mathbin{+\!\!+} w, o \rangle \to \langle i, k, w, \{u\} \mathbin{+\!\!+} o \rangle}$$

Where $\mathsf{insensitive}(t, k)$ means that $(= t'\ u) \in k \Rightarrow \neg\mathsf{unify}(t, t')$.

4 Ground literals and builtins

As with all practical programming languages, MeTTa hosts a number of computational entities and operations that are already defined on the vast majority of platforms on which an implementation of the language may be written and/or run. Here we describe the ground literals and builtin operations that every compliant MeTTa operation must provide.

4.1 Ground literals

As the grammar spells out every compliant implementation of MeTTa must provide: Booleans; signed and unsigned 64bit integers; 64bit floating point; strings.

4.2 Polymorphic operations

Every compliant implementation of the MeTTa client must provide a number of polymorphic operations including:

- $* : A \times A \to A$ for A ranging over Booleans, integers, floating point;
- $+ : A \times A \to A$ for A ranging over Booleans, integers, floating point, and strings;

4.3 Transition rules

As these rules are more or less standard and do not illustrate what is unique about MeTTa we give only a single example of how they are expressed. The interested reader is referred to the full semantics.

BoolAdd1

$$\langle \{(+\ b_1\ b_2)\} \mathbin{+\!\!+} i, k, w, o \rangle \to \langle i, k, w, \{b_1 || b_2\} \mathbin{+\!\!+} o \rangle$$

BoolAdd2

$$\frac{w = \{(+\ b_1\ b_2)\} \mathbin{+\!\!+} w'}{\langle i, k, w, o \rangle \to \langle i, k, w', \{b_1 || b_2\} \mathbin{+\!\!+} o \rangle}$$

5 Bisimulation

Since the operational semantics is expressed as a transition system we recover a notion of bisimulation. There are two possible ways to generate the notion of bisimulation in this context. One uses the Leifer-Milner-Sewell approach of deriving a bisimulation from the rewrite rules [4]. However, the technical apparatus is very heavy to work with. The other is to adapt barbed bisimulation developed for the asynchronous π-calculus to this setting.

The reason we need to use some care in developing the notion of bisimulation is that there are substitutions being generated and applied in many of the rules. So, a single label will not suffice. However, taking a query in the input space as a barb will. This notion of barbed bisimulation will provide a means of evaluating the correctness of compilation schemes to other languages. We illustrate this idea in the section on compiling MeTTa to the rho-calculus.

6 Compiling MeTTa to rho

In this section we illustrate the value of having an operational semantics by developing a compiler from MeTTa to the rho-calculus, and resource-bounded MeTTa to rholang. The essence of the translation is to use a channels for each of the registers.

6.1 MeTTa to the rho-calculus

In the translation given below we employ two semantic functions. One, written $[\![-]\!]_C$, turns a state into a collection of data deployed at four channels, i, k, w, o, and another, written $[\![-]\!]_E$, that processes that data. We also employ a semantic function, written $[\![-]\!]$,that transliterates MeTTa term syntax into rho term syntax for use in pattern matching, etc.

The astute reader will notice that there is a question about the well-definedness of $[\![-]\!]_E$. It is possible that there are multiple transitions out of a single state. The actual function takes a sum of all possible transitions:

$$[\![S]\!]_E := \Sigma_{r \in \{r | S \xrightarrow{r} S'\}} [\![S]\!]_r$$

where $[\![-]\!]_r$ are defined below.

The meaning of a MeTTa computation is given as the composition of the configuration and evaluation functions. That is,

$$[\![\langle i, k, w, o\rangle]\!]_M = [\![\langle i, k, w, o\rangle]\!]_C \mid [\![\langle i, k, w, o\rangle]\!]_E$$

The reason for this factorization of the semantics is to facilitate the proof of correctness, as we will see in the proof.

Space configuration

$$\begin{aligned}
[\![\langle \{t\} \mathbin{+\!\!+} i, k, w, o\rangle]\!]_C(i, k, w, o) &= i!([\![t]\!]) \mid [\![\langle i, k, w, o\rangle]\!]_C(i, k, w, o) \\
[\![\langle i, \{t\} \mathbin{+\!\!+} k, w, o\rangle]\!]_C(i, k, w, o) &= k!([\![t]\!]) \mid [\![\langle i, k, w, o\rangle]\!]_C(i, k, w, o) \\
[\![\langle i, k, \{t\} \mathbin{+\!\!+} w, o\rangle]\!]_C(i, k, w, o) &= w!([\![t]\!]) \mid [\![\langle i, k, w, o\rangle]\!]_C(i, k, w, o) \\
[\![\langle i, k, w, \{t\} \mathbin{+\!\!+} o\rangle]\!]_C(i, k, w, o) &= o!([\![t]\!]) \mid [\![\langle i, k, w, o\rangle]\!]_C(i, k, w, o)
\end{aligned}$$

Space evaluation

$[\![\langle\{t'\} \texttt{++}\ i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \texttt{++}\ k, w, o\rangle]\!]_{Query}(i,k,w,o)$

$=$

$\texttt{for}([\![t']\!] \leftarrow i)\{(\texttt{new}\ s)\{\Pi_{j=1}^{n}\texttt{for}((= [\![t']\!]\ v) \leftarrow k)\{w!([\![u_j]\!])|s!((= [\![t']\!]\ v))\}$
$\quad | \texttt{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{$
$\quad\quad \Pi_{j=1}^{n}k!(r_j) \mid [\![\langle i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \texttt{++}\ k, w, o\rangle]\!]_E(i,k,w,o)$
$\quad \}\}\}$

$[\![\langle i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \texttt{++}\ k, \{u\} \texttt{++}\ w, o\rangle]\!]_{Chain}(i,k,w,o)$

$=$

$\texttt{for}([\![u]\!] \leftarrow o)\{(\texttt{new}\ s)\{\Pi_{j=1}^{n}\texttt{for}(((= [\![u]\!]\ v) \leftarrow k)\{w!([\![u_j]\!])|s!((= [\![u]\!]\ v))\}$
$\quad | \texttt{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{$
$\quad\quad \Pi_{j=1}^{n}k!(r_j) \mid [\![\langle i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \texttt{++}\ k, w, o\rangle]\!]_E(i,k,w,o)$
$\quad \}\}\}$

$[\![\langle\{(\texttt{transform}\ t\ u)\} \texttt{++}\ i, \{t_1, \ldots, t_n\} \texttt{++}\ k, w, o\rangle]\!]_{Transform}(i,k,w,o)$

$=$

$\texttt{for}((\texttt{transform}\ [\![t]\!]\ [\![u]\!]) \leftarrow i)\{(\texttt{new}\ s)\{\Pi_{j=1}^{n}\texttt{for}([\![t]\!] \leftarrow k)\{w!([\![u]\!])|s!([\![t]\!])\}$
$\quad | \texttt{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{$
$\quad\quad \Pi_{j=1}^{n}k!(r_j) \mid [\![\langle i, \{t_1, \ldots, t_n\} \texttt{++}\ k, w, o\rangle]\!]_E(i,k,w,o)$
$\quad \}\}\}$

$[\![\langle\{(\texttt{addAtom}\ t)\} \texttt{++}\ i, k, w, o\rangle]\!]_{AddAtom1}(i,k,w,o)$

$=$

$\texttt{for}((\texttt{addAtom}\ [\![t]\!]) \leftarrow i)\{k!([\![t]\!]) \mid [\![\langle i, k, w, o\rangle]\!]_E(i,k,w,o)\}$

$[\![\langle i, \{(\texttt{addAtom}\ t)\} \texttt{++}\ k, w, o\rangle]\!]_{AddAtom2}(i,k,w,o)$

$=$

$\texttt{for}((\texttt{addAtom}\ [\![t]\!]) \leftarrow k)\{k!([\![t]\!]) \mid [\![\langle i, \{(\texttt{addAtom}\ t)\} \texttt{++}\ k, w, o\rangle]\!]_E(i,k,w,o)\}$

$[\![\langle\{(\texttt{remAtom}\ t)\} \texttt{++}\ i, \{t\} \texttt{++}\ k, w, o\rangle]\!]_{RemAtom1}(i,k,w,o)$

$=$

$\texttt{for}((\texttt{remAtom}\ [\![t]\!]) \leftarrow i)\{\texttt{for}([\![t]\!] \leftarrow k)\{o!([\![()]\!])\} \mid [\![\langle i, k, w, o\rangle]\!]_E(i,k,w,o)\}$

$[\![\langle i, \{(\texttt{remAtom}\ t)\} \texttt{++}\ \{t\} \texttt{++}\ k, w, o\rangle]\!]_{RemAtom2}(i,k,w,o)$

$=$

$\texttt{for}((\texttt{remAtom}\ [\![t]\!]) \leftarrow k)\{\texttt{for}([\![t]\!] \leftarrow k)\{o!([\![()]\!])\} \mid [\![\langle i, \{(\texttt{remAtom}\ t)\} \texttt{++}\ k, w, o\rangle]\!]_E(i,k,w,o)\}$

6.2 Correctness of the translation

Theorem 1 (MeTTa2rho correctness).

$$S_1 \dot{\approx} S_2 \iff [\![S_1]\!]_M \dot{\approx} [\![S_2]\!]_M$$

Proof. Proof sketch: Essentially, the translation is correct-by-construction. Intuitively, we see that there is a bisimulation relation between MeTTa computations and their translations in the rho-calculus. This bisimulation may be composed with a bisimulation between MeTTa calculations to yield a bisimulation in the rho-translation, and vice versa. The bisimulation bridging between the two domains is effectively represented in the translation. For each different kind of state, the left hand side of each bisimulation pair is the left hand side of the *definition* evaluation semantic function and the right hand side of the bisimulation pair is the right hand side of the definition. Hence, correct-by-construction. The formal proof uses terms in the input, working, and output registers as barbs for the notion of bisimulation on MeTTa computations, while their translations via the configuration function are barbs in the bisimulation in the rho-calculus.

7 Conclusion and future work

We have presented two versions of the operational semantics for MeTTa, one that is fit for private implementations that have some external security model, and one that is fit for running in a decentralized setting.

This semantics does not address typed versions of MeTTa. An interesting avenue of approach is the apply Meredith and Stay's OSLF to this semantics to derive a type system for MeTTa that includes spatial and behavioral types [10].

References

1. Gergely Buday. Formalising the SECD machine with nominal isabelle. In *SAC*, pages 1823–1824. ACM, 2015.
2. Ben Goertzel, Matt Ikle, and Alexey Potapov. Hyperon as agi approach, Nov 2022.
3. David Hart and Ben Goertzel. Opencog: A software framework for integrative artificial general intelligence. In *AGI*, volume 171 of *Frontiers in Artificial Intelligence and Applications*, pages 468–472. IOS Press, 2008.
4. James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2000.
5. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, February 2012.
6. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.
7. Lucius Gregory Meredith, Mike Stay, and Joshy Orndorff. Rholang tutorial, Mar 2021.
8. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
9. Gordon D. Plotkin. The origins of structural operational semantics. In *Journal of Logic and Algebraic Programming*, pages 60–61, 2004.
10. Christian Williams and Michael Stay. Native type theory. *CoRR*, abs/2102.04672, 2021.
11. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *GitHUB*, 2014.