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.

1 Introduction and motivation

We present the operational semantics for the language MeTTa. MeTTa is designed as a language in which humans and AGIs write the behavior of AGIs. One of the principle motivations of this document is to help developers of MeTTa clients know what is a correct and compliant implementation. The document serves roughly the same function as the JVM specification or Ethereum's Yellow paper.

2 Towards a common language for computational dynamics

Three of the most successful branches of scientific discourse all agree on the shape of a model adequate for expressing and effecting computation. Physics, computer science, and mathematics all use the same standard shape. A model adequate for computation comes with an algebra of states and "laws of motion."

One paradigmatic example from physics is Hilbert spaces and the Schroedinger equation. In computer science and mathematics the algebra of states is further broken down into a monad (the free algebra of states) and an algebra of the monad recorded as some equations on the free algebra.

Computer science represents laws of motion, aka state transitions, as rewrite rules exploiting the structure of states to determine transitions to new states. Mainstream mathematics is a more recognizable generalization of physics, coding state transitions, aka behavior, via morphisms (including automorphisms) between state spaces.

But all three agree to a high degree of specificity on what ingredients go into a formal presentation adequate for effecting computation.

2.1 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.

λ-calculus

Algebra of States

$$\begin{aligned} Term[V] ::= & \ V \\ | & \ \lambda V.Term[V] \\ | & \ (Term[V]\ Term[V]) \end{aligned}$$

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.

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

π-calculus

Algebra of States

$$\begin{aligned} Term[N] ::= & \ \mathsf{0} \\ | & \ \mathsf{for}(N \leftarrow N)Term[N] \\ | & \ N!(N) \\ | & \ (\mathsf{new}\ N)Term[N] \\ | & \ Term[N] \mid Term[N] \\ | & \ !Term[N] \end{aligned}$$

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

$$\begin{aligned} &(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}\ x)P \\ &(\mathsf{new}\ x)(\mathsf{new}\ y)P \equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P \\ &\quad ((\mathsf{new}\ x)P)|Q \equiv (\mathsf{new}\ x)(P|Q), x \notin \mathsf{FN}(Q) \end{aligned}$$

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.

$$\text{COMM}\quad \mathsf{for}(y \leftarrow x)P | x!(z) \rightarrow P\{z/y\}$$

$$\text{PAR}\quad \frac{P \rightarrow P'}{(\mathsf{new}\ x)P \rightarrow (\mathsf{new}\ x)P'} \qquad \text{PAR}\quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{STRUCT}\quad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q}$$

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 $Z = \{0\}$.

$$\text{PROCESS}\quad \begin{aligned} Term[Z] ::= Z \ \mid\ & \mathsf{for}(Name[Z] \leftarrow Name[Z])Term[Z] \ \mid\ Name[Z]!(Term[Z]) \\ & \mid *Name[Z] \ \mid\ Term[Z]|Term[Z] \end{aligned}$$

$$\text{NAME}\quad Name[Z] ::= @Term[Z]$$

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

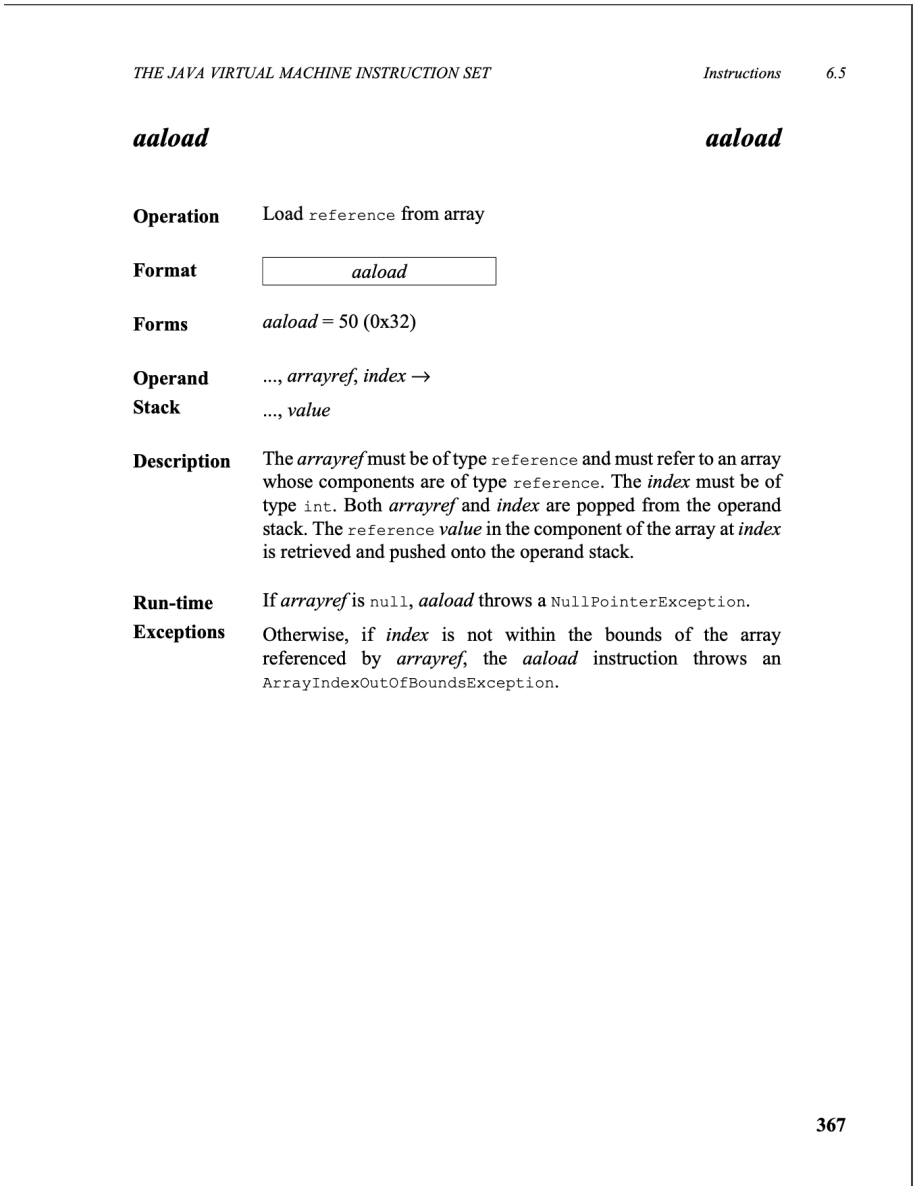
Again, it is evident that $Term[Z]$ is a monad and imposing the structural congruence gives an algebra of the monad.

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

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

$$\text{PAR}\quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \text{STRUCT}\quad \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

The JVM While its complexity far exceeds the presentations above, the JVM specification respects this same shape. Here is an example from the specification of what the operation aaload does.



THE JAVA VIRTUAL MACHINE INSTRUCTION SET *Instructions* *6.5*

aaload ***aaload***

Operation Load `reference` from array

Format | *aaload* |

Forms *aaload* = 50 (0x32)

Operand Stack ..., *arrayref*, *index* → ..., *value*

Description The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The `reference` *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.

Run-time Exceptions If *arrayref* is `null`, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

367

Fig. 1. AALOAD instruction specification

Register machines and WYSIWYG semantics One important point about the JVM versus the previous three examples. The first three examples are examples of WYSIWYG operational semantics in the sense that the states *are* the terms of the calculi. In the case of the JVM the terms in the language are only part of the state, which includes the stack,

the heap, and several registers. WYSIWYG models make static analysis dramatically simpler. Specifically, an analyzer only has to look at terms in the language.

This phenomenon is not restricted to the JVM. Even the famous SECD machine is not strictly WYSIWYG. In fact, the register based states are not strictly a monad, but contain a monad.

3 A presentation of the semantics of MeTTa

A presentation of the semantics of MeTTa must therefore provide a monad describing the algebra of states, a structural equivalence quotienting the algebra of states, and some rewrite rules describing state transitions. Such a description is the minimal description that meets the standard for describing models of computation.

Note that to present such a description requires at least that much expressive power in the system used to formalize the presentation. That is, the system used to present a model of computation is itself a model of computation admitting a presentation in terms of an algebra of states and some rewrites. This is why a meta-circular evaluator is a perfectly legitimate presentation. That is, a presentation of MeTTa's semantics in MeTTa is perfectly legitimate. Meta-circular presentations are more difficult to unpack, which is why such presentations are typically eschewed, but they are admissible. In fact, a meta-circular evaluator may be the most pure form of presentation.

But, this fact has an important consequence. No model that is at least Turing complete can be "lower level" than any other.

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.

One of the motivations for providing the example to is foster within the MeTTa developer community a development methodology known as correct-by-construction. Some of the benefits of correct-by-construction include avoiding bytecode injection attacks, avoiding concurrency issues, and in general avoiding technical debt.

With a clearly spelled out operational semantics the correct-by-construction software development methodology becomes a real practice and not just an ideal to be striven for – except when we're under deadling, and we're always under deadline!

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. 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.

In short, scaling is not just about performance and throughput; it's also about adoption. Adoption at scale is a very error prone process, as anyone familiar with the histories of a wide range of computer-based technologies will attest. Linux users, for example, will bear witness to the historical lag between device drivers for Windows and Mac versus the drivers compliant to the same specs that ran on Linux. Correct-by-construction dramatically reduces the number of errors in multiple independent implementations, and thus makes scaling through adoption a much more tractable proposition.

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

$$
\begin{aligned}
Term ::= {} & (Term\ [Term]) \\
| \ & \{Term\ [Term]\} \\
| \ & (Term \mid [Receipt]\ .\ [Term]) \\
| \ & \{Term \mid [Receipt]\ .\ [Term]\} \\
| \ & Atom
\end{aligned}
$$

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.

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.

Extensional vs intensionally defined spaces We make 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.

We adopt this design 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.

Explicit bindings Comprehensions provide a superior framework for the explicit representation of bindings. For example, they significantly generalize let and letrec constructs. In particular, the generally accepted semantics for these constructs do not extend smoothly to streams.

Infinite terms and spaces In fact, since the advent of set comprehensions and continuing through SQL's SELECT – FROM – WHERE to Haskell's do-nation and Scala's for-comprehensions the general mechanism for describing intensionally specified collections has been shown to be a powerful abstraction mechanism. Specifically, we now know that comprehensions are really syntactic sugar for the monadic operations, which makes these an exceptionally flexible framework for representing a notion of binding across a wide range of computational phenomena.

Remotely accessed data Whether access data from a distributed atom space, or a resource on the Internet, or calling a foreign function across a memory boundary, remotely accessed data comes with different failure modes. Data providers can be offline or otherwise inaccessible. Data can be ill-formatted or peppered with an array of challenges, from buffer or register overflows to triggering divergent computation.

Providing a scope for these failure modes has a distinct advantage for defensive computation. Beyond that, however, are issues related with fair merge of divergent behavior. The famous example from PCF was logical disjunction. An evaluation strategy for disjunction that evaluates both arguments will diverge if one of the arguments diverges. An evaluation that returns true immediately if the first argument it evaluates is true will potentially diverge less often. This generalizes to a wide range of situations involving the integration of multiple foreign sources of data.

While there is not a one-size-fits all solution, Oleg Kiselyov has provided a natural mechanism in the monad transformer, LogicT. This provides a policy language for describing merge policies. It is an elegant solution that fits perfectly with comprehensions.

Term sequences

$$
\begin{aligned}
[Term] ::= {} & \epsilon \\
| \ & Term \\
| \ & Term\ [Term]
\end{aligned}
$$

Bindings

$$
\begin{aligned}
Receipt ::= {} & ReceiptLinearImpl \\
| \ & ReceiptRepeatedImpl \\
| \ & ReceiptPeekImpl
\end{aligned}
$$

$$
\begin{aligned}
[Receipt] ::= {} & Receipt \\
| \ & Receipt;\ [Receipt]
\end{aligned}
$$

$$
\begin{aligned}
ReceiptLinearImpl ::= {} & [LinearBind] \\
LinearBind ::= {} & [Name]\ NameRemainder \leftarrow AtomSource
\end{aligned}
$$

$$
\begin{aligned}
[LinearBind] ::= {} & LinearBind \\
| \ & LinearBind\ \&\ [LinearBind]
\end{aligned}
$$

$$
\begin{aligned}
AtomSource ::= {} & Name \\
| \ & Name?! \\
| \ & Name!?([Term])
\end{aligned}
$$

$$
\begin{aligned}
ReceiptRepeatedImpl ::= {} & [RepeatedBind] \\
RepeatedBind ::= {} & [Name]\ NameRemainder \Leftarrow Atom
\end{aligned}
$$

$$
\begin{aligned}
[RepeatedBind] ::= {} & RepeatedBind \\
| \ & RepeatedBind\ \&\ [RepeatedBind]
\end{aligned}
$$

$$
\begin{aligned}
ReceiptPeekImpl ::= {} & [PeekBind] \\
PeekBind ::= {} & [Name]\ NameRemainder \leftharpoonup Atom
\end{aligned}
$$

$$
\begin{aligned}
[PeekBind] &::= PeekBind \\
&\mid PeekBind \,\&\, [PeekBind]
\end{aligned}
$$

$$
\begin{aligned}
TermRemainder &::= \ldots TermVar \\
&\mid \epsilon \\
NameRemainder &::= \ldots @TermVar \\
&\mid \epsilon
\end{aligned}
$$

Literals and builtins

$$
\begin{aligned}
Atom &::= Ground \\
&\mid Builtin \\
&\mid Var \\
Name &::= \_ \\
&\mid Var \\
&\mid @Term
\end{aligned}
$$

$$
\begin{aligned}
[Name] &::= \epsilon \\
&\mid Name \\
&\mid Name,[Name] \\
BoolLiteral &::= \mathsf{true} \\
&\mid \mathsf{false} \\
Ground &::= BoolLiteral \\
&\mid LongLiteral \\
&\mid StringLiteral \\
&\mid UriLiteral \\
Builtin &::= ::= \\
&\mid = \\
TermVar &::= \_ \\
&\mid Var
\end{aligned}
$$

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 the following polymorphic operations:

- $* : A \times A \to A$ 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

$$\textsc{BoolAdd1}\quad \langle \{(+\ b_1\ b_2)\} \mathbin{+\!\!+} i, k, o\rangle \to \langle i, k, \{b_1 || b_2\} \mathbin{+\!\!+} o\rangle$$

$$\textsc{BoolAdd2}\quad \frac{k = \{(+\ b_1\ b_2)\} \mathbin{+\!\!+} k'}{\langle i, k, o\rangle \to \langle i, k, \{b_1 || b_2\} \mathbin{+\!\!+} o\rangle}$$

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. 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 The cost of transitions

6.1 Network access tokens

If you're reading this, chances are that you know what an Internet-facing API is, and why it might need to be protected from denial of service attacks. But, just in case you're one of the "normies" that don't know what these terms refer to, let's you, me, Sherman, and Mr. Peabody all take a trip in the WayBack Machine way back to 2005.

In those days there was still a naivete about the infinite potential of free and open information. QAnon, deep fakes, ChatGPT and other intimations that the Internet might just be the modern equivalent of the Tower of Babel were not yet even a gleam in their inventors' eyes. Companies would regularly set up network services that anyone with an Internet connection could access, from anywhere in the world (dubbed Internet-facing). Such services were accessed by sending requests in a particular, well defined format (deriving from the software term application program interface, or API) to an Internet address served by machines in the network service the organization had set up.

It was quickly discovered that such Internet-facing APIs were vulnerable to attack. If a single bad actor sent thousands or millions of requests to the service, or a botnet of millions sent a few requests each to the service, it was possible for the service to become bogged down and unresponsive to legitimate requests. Now, in reality, all this was discovered long before 2005. But, by 2005 a practice for dealing with this kind of attack was more or less well established.

The solution is simple. The network proprietor issues a digital token. A request with a given token embedded in it is honored, up to some number of requests per token. This practice is less onerous and costly than having to issue and maintain authorization credentials for login challenges. Many, many companies do this and have done this for the better part of two decades. Not just software or digital service companies like Google and Microsoft issue tokens like this, Other companies, such as media companies like The New York Times, and The Guardian, also employ this practice. (The hyperlinks above are to their token distribution pages.) The practice is ubiquitous and well accepted. It is intrinsic to the functionality of an open network such as the Web.

Also, it is important to note that many of these services allow for storage of digital content on the networks provided by these services. However, bad actors can still abuse the services by repeatedly uploading illegal content (like child pornography, copyrighted material or even nuclear secrets). So, an entity offering Internet-enabled services must reserve the right to invalidate these tokens in case they discover they are being abused in this or other ways. These utility tokens are essential to comply with a whole host of very good laws.

6.2 Ethereum's big idea

Satoshi's discovery of a new class of economically secured, leaderless distributed consensus protocols, embodied in proof-of-work but also, elsewhere, embodied in proof-of-stake and other consensus algorithms, was a pretty good idea. It led to the Bitcoin network. Buterin's

suggestion that Satoshi's consensus be applied to the state of a virtual machine instead of a ledger was a really good idea, and led to the Ethereum network. It creates a distributed computer that runs everywhere and nowhere in particular. Less poetically, every node in the network is running a copy of the virtual machine and the consensus protocol ensures that all the copies agree on the state of the virtual machine.

Like the Internet-facing APIs launched all throughout the 00's and beyond, Ethereum's distributed computer is accessible to anyone with an Internet connection. And, as such, without protection would be vulnerable to denial of service attacks. In fact, it's potentially even more vulnerable because a request to the Ethereum distributed computer is a piece of code. This code could, in principle, run forever, or take up infinite storage space. Vitalik's clever idea, building on the established practice of network access tokens, is to require tokens for each computational or storage step to prevent such abuses.

6.3 MeTTa effort objects

MeTTa takes the same approach. Transitions in the operational semantics cost a computational resource (effort objects, or EOs, for short) that are "purchased" with tokens. This section reprises the operational semantics with the cost of each step spelled out in terms of the structure of EOs.

Resource-bounded Rewrite Rules We assume a polymorphic cost function # taking values in the domain of EOs. We assume the domain of EOs supports a notion of + making it a *commutative* monoid.

QUERY

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

CHAIN

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

TRANSFORM

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

ADDATOM1

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

ADDATOM2

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

REMATOM1

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

REMATOM2

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

OUTPUT

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

7 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.

7.1 MeTTa to the rho-calculus

Space configuration

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

Space evaluation

$$
\begin{aligned}
&[\![\langle \{t'\} \mathbin{+\!\!+} i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}([\![t']\!] \leftarrow i)\{(\mathsf{new}\ s)\{\Pi_{j=1}^{n}\mathsf{for}((= [\![t']\!]\ v) \leftarrow k)\{w!([\![u_j]\!])|s!((= [\![t']\!]\ v))\}\\
&\qquad\qquad \mid \mathsf{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{\Pi_{j=1}^{n}k!(r_j)\}\}\}\\
&[\![\langle i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k, \{u\} \mathbin{+\!\!+} w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}([\![u]\!] \leftarrow o)\{(\mathsf{new}\ s)\{\Pi_{j=1}^{n}\mathsf{for}(((= [\![u]\!]\ v) \leftarrow k)\{w!([\![u_j]\!])|s!((= [\![u]\!]\ v))\}\\
&\qquad\qquad \mid \mathsf{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{\Pi_{j=1}^{n}k!(r_j)\}\}\}\\
&[\![\langle \{(\mathsf{transform}\ t\ u)\} \mathbin{+\!\!+} i, \{t_1, \ldots, t_n\} \mathbin{+\!\!+} k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}((\mathsf{transform}\ [\![t]\!]\ [\![u]\!]) \leftarrow i)\{(\mathsf{new}\ s)\{\Pi_{j=1}^{n}\mathsf{for}([\![t]\!] \leftarrow k)\{w!([\![u]\!])|s!([\![t]\!])\}\\
&\qquad\qquad\qquad\qquad \mid \mathsf{for}(r_1 \leftarrow s\ \&\ \ldots\ \&\ r_n \leftarrow s)\{\Pi_{j=1}^{n}k!(r_j)\}\}\}\\
&[\![\langle \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} i, k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}((\mathsf{addAtom}\ [\![t]\!]) \leftarrow i)\{k!([\![t]\!])\}\\
&[\![\langle i, \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}((\mathsf{addAtom}\ [\![t]\!]) \leftarrow k)\{k!((\mathsf{addAtom}\ [\![t]\!]))|k!([\![t]\!])\}\\
&[\![\langle \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} i, \{t\} \mathbin{+\!\!+} k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}((\mathsf{remAtom}\ [\![t]\!]) \leftarrow i)\{\mathsf{for}([\![t]\!] \leftarrow k)\{o!([\![()]\!])\}\}\\
&[\![\langle i, \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} \{t\} \mathbin{+\!\!+} k, w, o\rangle]\!](i,k,w,o)\\
&=\\
&\mathsf{for}((\mathsf{remAtom}\ [\![t]\!]) \leftarrow k)\{\mathsf{for}([\![t]\!] \leftarrow k)\{k!((\mathsf{remAtom}\ [\![t]\!]))|o!([\![()]\!])\}\}
\end{aligned}
$$

7.2 Correctness of the translation

Theorem 1 (MeTTa2rho correctness).

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

Proof. Proof sketch: the barbs are terms in the input, working, and output registers.

7.3 Resource-bounded MeTTa to rholang

$$\begin{aligned}
[\![\langle \{t'\} \mathbin{+\!\!+} i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k, o; eos\rangle]\!] &= ??? \\
[\![\langle i, \{(= t_1\ u_1), \ldots, (= t_n\ u_n)\} \mathbin{+\!\!+} k, \{u\} \mathbin{+\!\!+} o; eos\rangle]\!] &= ??? \\
[\![\langle \{(\mathsf{transform}\ t\ u)\} \mathbin{+\!\!+} i, \{t_1, \ldots, t_n\} \mathbin{+\!\!+} k, o; eos\rangle]\!] &= ??? \\
[\![\langle \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} i, k, o; eos\rangle]\!] &= ??? \\
[\![\langle i, \{(\mathsf{addAtom}\ t)\} \mathbin{+\!\!+} k, o; eos\rangle]\!] &= ??? \\
[\![\langle \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} i, \{t\} \mathbin{+\!\!+} k, o; eos\rangle]\!] &= ??? \\
[\![\langle i, \{(\mathsf{remAtom}\ t)\} \mathbin{+\!\!+} \{t\} \mathbin{+\!\!+} k, o; eos\rangle]\!] &= ???
\end{aligned}$$

7.4 Correctness of the translation

Theorem 2 (MeTTa2rho correctness).

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

8 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 addressed 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.

References

1. Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL*, pages 33–44. ACM, 2002.
2. Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.
3. Samson Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.
4. Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.

5. Luis Caires. Spatial logic model checker, Nov 2004.
6. Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Inf. Comput.*, 186(2):194–235, 2003.
7. Luís Caires and Luca Cardelli. A spatial logic for concurrency - II. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
8. Luca Cardelli. Brane calculi. In *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2004.
9. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.
10. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002.
11. Cédric Fournet, Georges Gontheir, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. Allen L. Brown Jr., Cosimo Laneve, and L. Gregory Meredith. Piduce: A process calculus with native XML datatypes. In *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2005.
14. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
15. Greg Meredith. Documents as processes: A unification of the entire web service stack. In *WISE*, pages 17–20. IEEE Computer Society, 2003.
16. L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *TGC* [17], pages 353–369.
17. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.
18. Lucius Meredith, Jan 2017.
19. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
20. Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
21. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
22. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.
23. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.
24. Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.
25. Davide Sangiorgi. Beyond bisimulation: The "up-to" techniques. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2005.
26. Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.
27. Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1992.
28. Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32(4):12:1–12:63, 2010.