Meta-MeTTa: an operational semantics for MeTTa

Lucius Gregory Meredith[1]
and Ben Goertzel[2]

[1] CEO, F1R3FLY.io 9336 California Ave SW, Seattle, WA 98103, USA
`f1r3fly.ceo.com`
[2] CEO, SingularityNet
`ben@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 human 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 and a structural congruence, and the rewrite rules in Plotkin-style SOS format.

λ-calculus

Algebra of States

$$Term[V] ::= V$$
$$| \quad \lambda V.Term[V]$$
$$| \quad (Term[V]\, Term[V])$$

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.

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

$$\text{BETA}$$
$$((\lambda x.M)N) \to M\{N/x\}$$

π-calculus

Algebra of States

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

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

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

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

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

$$\text{PAR}$$
$$\frac{P \to P'}{P|Q \to P'|Q}$$

$$\text{STRUCT}$$
$$\frac{P \equiv P' \to Q' \equiv Q}{P \to Q}$$

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.

PROCESS
$$P, Q ::= 0 \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid {*}x \mid P|Q$$

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

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

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

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

Fig. 1. AALOAD instruction specification

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.

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 implementations;
- meta-level computation, including type checking, model checking, macros, computational reflection, etc.

3.2   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.

Algebra of States

Terms

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

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.

Term sequences

$$[Term] ::= \epsilon$$
$$| \quad Term$$
$$| \quad Term \, [Term]$$

Bindings

$$Receipt ::= Receipt LinearImpl$$
$$| \quad Receipt RepeatedImpl$$
$$| \quad Receipt PeekImpl$$

$$[Receipt] ::= Receipt$$
$$| \quad Receipt; [Receipt]$$

$$ReceiptLinearImpl ::= [LinearBind]$$
$$LinearBind ::= [Name] \, NameRemainder \leftarrow AtomSource$$

$$[LinearBind] ::= LinearBind$$
$$| \quad LinearBind \, \& \, [LinearBind]$$

$$AtomSource ::= Name$$
$$| \quad Name?!$$
$$| \quad Name!?([Term])$$
$$ReceiptRepeatedImpl ::= [RepeatedBind]$$
$$RepeatedBind ::= [Name] \, NameRemainder \Leftarrow Atom$$

$$[RepeatedBind] ::= RepeatedBind$$
$$| \quad RepeatedBind \, \& \, [RepeatedBind]$$
$$ReceiptPeekImpl ::= [PeekBind]$$
$$PeekBind ::= [Name] \, NameRemainder \hookleftarrow Atom$$

$$[PeekBind] ::= PeekBind$$
$$| \quad PeekBind \;\&\; [PeekBind]$$

$$TermRemainder ::= ...\, TermVar$$
$$| \quad \epsilon$$
$$NameRemainder ::= ...\, @TermVar$$
$$| \quad \epsilon$$

Literals and builtins

$$Atom ::= Ground$$
$$| \quad Builtin$$
$$| \quad Var$$
$$Name ::= \_$$
$$| \quad Var$$
$$| \quad @Term$$

$$[Name] ::= \epsilon$$
$$| \quad Name$$
$$| \quad Name,[Name]$$
$$BoolLiteral ::= \mathsf{true}$$
$$| \quad \mathsf{false}$$
$$Ground ::= BoolLiteral$$
$$| \quad LongLiteral$$
$$| \quad StringLiteral$$
$$| \quad UriLiteral$$
$$Builtin ::= ::=$$
$$| \quad =$$
$$| \quad :$$
$$TermVar ::= \_$$
$$| \quad Var$$

States

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

We will use S, T, U to range over states and $\mathsf{i} := \pi_1$, $\mathsf{k} := \pi_2$, and $\mathsf{o} := \pi_3$ for the first, second, and third 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. $t\sigma$.

Rewrite Rules

$$\sigma_i = \mathsf{unify}(t_i, t')$$
$$\overline{\langle \{t'\} \mathbin{+\!\!+} i, \{(= t_1\ u_1), \ldots, (= t_k\ u_k)\} \mathbin{+\!\!+} k, o\rangle \to \langle i, \{(= t_1\ u_1), \ldots, (= t_k\ u_k)\} \mathbin{+\!\!+} k, \{u_i\sigma_i\} \mathbin{+\!\!+} o\rangle}$$

Transform

$$\sigma_i = \mathsf{unify}(t_i, t')$$
$$\overline{\langle \{(\mathsf{transform}\ t\ u)\} \mathbin{+\!\!+} i, \{t_1, \ldots, t_k\} \mathbin{+\!\!+} k, o\rangle \to \langle i, \{t_1, \ldots, t_k\} \mathbin{+\!\!+} k, \{u\sigma_i\} \mathbin{+\!\!+} o\rangle}$$

AddAtom1

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

AddAtom2

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

4 The cost of transitions

4.1 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.

4.2 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.

4.3 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 "purchases" with tokens. This section reprises the operational semantics with the cost of each step spelled out in terms of the structure of EOs.

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.