

# The MeTTa calculus

L.G. Meredith<sup>1</sup>

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.

## 1 Introduction and motivation

In [2] 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 a 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.

PROCESS		NAME	
$P, Q ::= 0 \mid G \mid \text{for}(t \leftrightarrow x)P \mid x?P \mid *x \mid P Q$		$x, y ::= @P$	
TERM		ATOM	
$t, u ::= atom \mid ([t])$		$atom ::= x \mid P$	
GROUND		COLLECTION	
$G ::= \text{Bool} \mid \text{String} \mid \text{Int} \mid C$		$C ::= [[t]_{seq}] \mid (t, [t]_{seq}) \mid \text{Set}([t]_{seq}) \mid \{[t:t]\}$	

$$\begin{array}{c}
\text{EQUIV} \\
P|0 \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)R \\
\\
\text{ALPHA} \\
\frac{\text{occurs}(t, y)}{\text{for}(t \leftrightarrow x)P \equiv \text{for}(t\{z/y\} \leftrightarrow x)(P\{z/y\}) \text{ if } z \notin \text{FN}(P)} \\
\\
\text{COMM} \\
\frac{\sigma = \text{unify}(t, u)}{\text{for}(t \leftrightarrow x)P \mid \text{for}(u \leftrightarrow x)Q \rightarrow P\dot{\sigma}|Q\dot{\sigma}} \\
\\
\begin{array}{cc}
\text{PAR} & \text{EQUIV} \\
\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} & \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \\
\\
\text{REFL} \\
\frac{P \rightarrow P'}{x?P \rightarrow x!(P')}
\end{array}
\end{array}$$

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

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

$$\begin{array}{cc}
\text{MeTTa language} & \text{MeTTa calculus} \\
(\text{addAtom } \textit{space term}) & \text{for}(\textit{term} \leftrightarrow \textit{space})0 \\
(\text{remAtom } \textit{space term}) & \text{for}(\textit{term} \leftrightarrow \textit{space})0 \\
(? \textit{space term}) & \text{for}(\textit{term} \leftrightarrow \textit{space})0
\end{array}$$

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  $t \leftrightarrow \{ P \}$ . When it is tagged with a space, say  $x$ , i.e.  $\text{for}(t \leftrightarrow x)P$ , 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  $\text{space} : \text{@Proc} \rightarrow \text{Proc}$  by

$\text{space}(x) = \Pi_i \text{for}(t_i \leftrightarrow x) P_i$ . 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

PROCESS

$$P, Q ::= \dots \mid !P \mid \text{new } x \text{ in } \{ P \}$$

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

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

PROCESS

$$P, Q ::= \dots \mid \text{for}([\text{Join}])P$$

JOINS

$$[\text{Join}] ::= \text{Join} \mid \text{Join};[\text{Join}]$$

JOIN

$$\text{Join} ::= [\text{Query}]$$

QUERIES

$$[\text{Query}] ::= \text{Query} \mid \text{Query} \& [\text{Query}]$$

QUERY

$$\text{Query} ::= t \leftrightarrow x$$

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 <=> reviewer1 & true <=> reviewer2 & true <=> reviewer3
){
  // 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

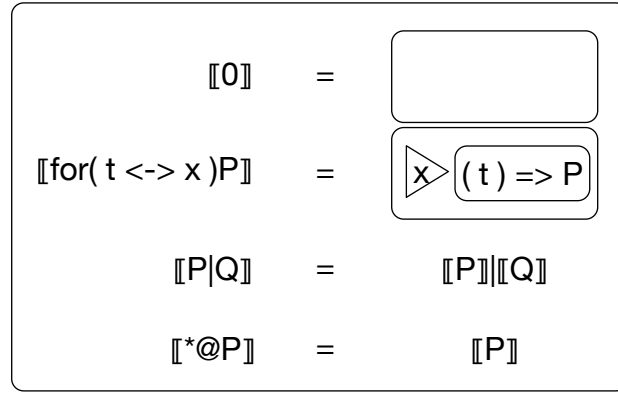
$$\text{COMM} \quad \frac{\sigma = \text{unify}(t, u)}{\text{for}(t \leftrightarrow x)P \mid \text{for}(u \leftrightarrow x)Q \rightarrow P\dot{\sigma} \mid Q\dot{\sigma}}$$

## 4 From calculus to efficient implementation

It turns out that a variant of the **Linda** tuple space [1] 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.

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**;



**Fig. 1.** Compiler from MeTTa-calculus to 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*; <sup>1</sup>
- 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.

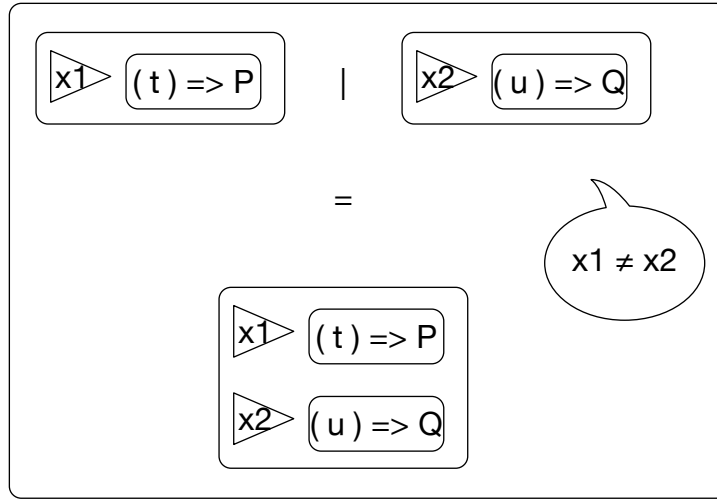
**Parallel composition of RSpaces** If we are combining two **RSpaces** that only have a single key-value pair, each; and the keys are not equal, then we simply combining them into a single **RSpace** containing both key-value pairs. More generally, when combining **RSpaces** that have no overlap in 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).

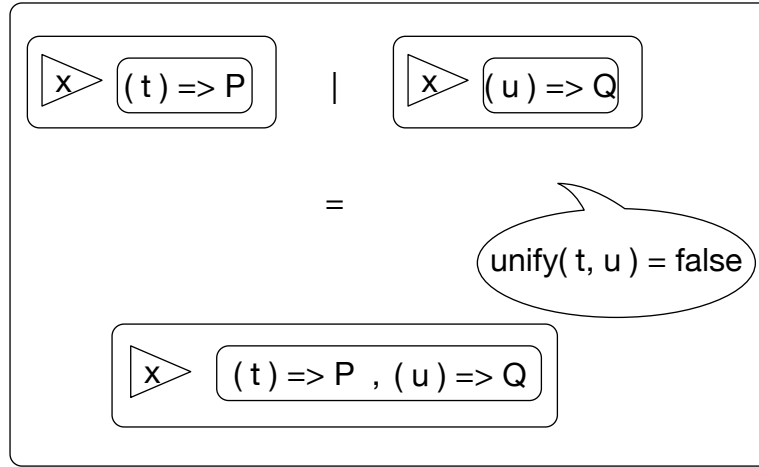
More generally, when combining **RSpaces** that have overlap in their key-sets, but the overlapping keys contain no continuations whose patterns unify, we simply return the **RSpace** whose key-value pairs is

- 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.

<sup>1</sup> Sometimes we abuse the terminology and call the body the continuation.



**Fig. 2.** Compiler from MeTTa-calculus to RSpace continued



**Fig. 3.** 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.]

## 5 Tokenized security

SECURITY-TOKENS

$T ::= () \mid s \mid T : T$

SECURED-PROCESSES

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

MULTI-PARTY-SIGS

$s ::= () \mid \text{hash}(<signature>) \mid s\&s$

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

$$\begin{array}{c}
\text{COMM-COSIGNED-PAR-EXTERNAL-SEQUENTIAL} \\
\frac{\sigma = \text{unify}(t, u)}{\{\text{for}(t \leftrightarrow x)P\}_{s_1} \mid \{\text{for}(u \leftrightarrow x)P\}_{s_2} \mid s_1 \& s_2 : T \rightarrow \{P\dot{\sigma} \mid Q\dot{\sigma}\}_{s_1 \& s_2} \mid T} \\
\\
\text{COMM-COSIGNED-PAR-EXTERNAL-CONCURRENT} \\
\frac{\sigma = \text{unify}(t, u)}{\{\text{for}(t \leftrightarrow x)P\}_{s_1} \mid \{\text{for}(u \leftrightarrow x)P\}_{s_2} \mid s_1 : T_1 \mid s_2 : T_1 \rightarrow \{P\dot{\sigma} \mid Q\dot{\sigma}\}_{s_1 \& s_2} \mid T_1 \mid T_2} \\
\\
\text{COMM-SIGNED} \\
\frac{P \rightarrow P'}{\{P\}_s \mid s : T \rightarrow \{P'\}_s \mid T} \\
\\
\text{COMM-COSIGNED-PAR-INTERNAL} \\
\frac{P \rightarrow P'}{\{P\}_{s_1 \& s_2} \mid s_1 : T_1 \mid s_2 : T_2 \rightarrow \{P'\}_s \mid T_1 \mid T_2}
\end{array}$$

## 6 Spatial-behavioral types

TBD

## 7 Stochastic and quantum execution

### 7.1 Stochastic execution

TBD

### 7.2 Quantum execution

TBD

## 8 Conclusions and future research

TBD

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

## References

1. David Gelernter. Multiple tuple spaces in linda. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 12-16, 1989, Proceedings*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer, 1989.
2. Adam Vandervorst Lucius Gregory Meredith, Ben Goertzel. Meta-metta: An operational semantics for metta, 2023.