MeTTa Architecture Proposal

Contents

1 Introduction

SingularityNet's investment in MeTTa is predicated on the idea that intelligence opportunistically creates theories of computation to suit the domain in which the intelligence is operating. In fact, the name MeTTa is intended as pun for a meta-level framework for specifying different theories of computation. SingularityNet hopes that MeTTa will be a language in which one can describe a theory of computation and get an efficient interpreter that is suited for smart contracts on a blockchain [4].

Without reproducing much of MeTTa's internals explicitly in the theory, MeTTa currently cannot model different evaluation strategies for lambda calculus, cannot model the semantics of the π calculus and other mobile process calculi, and has several other problems preventing it from living up to this vision. We propose an architecture based on category theoretical concepts that will enable MeTTa to:

- specify the semantics of any discrete theory of computation
- generate an efficient interpreter for that theory
- generate a sound type system customized to that theory
- support concurrency
- support transactions

and more.

Section 3.1 attempts to express some theories of computation in MeTTa and uncovers some problems. Section 3.2 describes graph-structured lambda theories (GSLTs), which solve the expressability problems. Section 3.3 shows how from a GSLT we can generate an efficient interpreter and a sound type system. The resulting compiled language is concurrent and transactional. Section **??** returns to the need to query a fact database and examines the kinds of interesting query one can express in this system.

1.1 Decentralization

Were the deployment of an execution environment for MeTTa SingularityNet's only goal the MeTTaCycle architecture would comprise the elements above. However, SingularityNet's goals include a decentralized framework for AI, AGI, and ASI. As such, the architecture must also address a security model for deployment of execution outside of the protective womb of a firewall. Further, it must embrace integration with a number of third party services. Despite these additional constraints and their corresponding remedies, MeTTaCycle must also be accessible and not hidden behind arcane or cryptic command line interfaces.

To achieve the former we adopt a capabilities driven tokenization model. Access to compute and storage are mediated by (object) capabilities represented as tokens. See the 4.1 for details. To achieve the latter we provide integration of MeTTaCycle with the file systems for the major operating systems; and, with major social media platforms. See 6.

Finally, to achieve decentralization the MeTTaCycle architecture needs to include fault-tolerance balanced against the need to replicate data everywhere. It achieves this by delivering a sharded, heterogeneous consensus mechanism. Each shard represents a unit of fault-tolerance governed by a consensus mechanism. Shards may be organized into a tree. Parent shards delegate to children when transactions only involve resources confined to a single child shard, but provide a governing consensus for transactions that implicate resources spanning sibling shards.

2 Market requirements

The demand for decentralized digital asset management systems is on the rise. By digital assets we don't merely mean "tokens" as used in BTC or ETH. Rather, we mean digitally recorded and stored information from medical records to code (such as is found in GitHub); and we expand this scope to agentic services that act on the behalf of their users. The MeTTaCycle architecture provides such a digital asset management service that guarantees the security model of Byzantine fault tolerant networks (BTC, ETH) with the throughput and scale of crash fault tolerant networks (Google Drive, Dropbox). This makes it dramatically more secure and useful than any of the incumbent platforms.

While the crash fault tolerant cloud-based networks have scaled to serve millions of concurrent active users, they have been plagued by hacks like Snowflake, Crowdstrike, and many others. By contrast, the Byzantine fault tolerant networks (such as BTC) have huge unclaimed "bug bounties". $ Bs of assets have been flowing through the BTC network for almost two decades without a successful hack that would allow such a claim to be made. Nevertheless, although the latter networks benefit from using Byzantine fault tolerant technology, they are in themselves unable to scale – a significant limitation. MeTTaCycle scales to match the throughput characteristics of the crash fault tolerant networks (CFTNs) and does this while also supplying the security of the Byzantine fault tolerant networks (BFTNs).

Furthermore, one of the crucial features of the CFTNs is their searchability. One of the most important features enabling the new global economy is search. When everyone with an Internet connection became able to search – for jobs or employees, goods, services, news, analysis, etc. – the global digital market became a reality, supplanting oil as the most valuable sector. MeTTaCycle brings the searchability of the CFTNs to the BFTNs. This not only means that data stored on MeTTaCycle networks can be searched. It also means it can be fed to and integrated with emerging AI solutions. Specifically, MeTTaCycle's smart contracting language is, by design, a transactional query language that

works across an entire MeTTaCycle network at scale, on the one hand; and a state of the art AI language on the other.

Unlike the BFTNs where tokenization seems to be primarily to encourage trading, the key role of tokenization in MeTTaCycle networks is fractional security. A token is a key to a unit of compute or a unit of storage. Imagine if, rather than handing over your Netflix password to your family members, you gave them an allotment of tokens, each one representing a prescribed number of viewing hours. That's what MeTTaCycle tokens represent: a very fine-grained security mechanism that enables careful accounting and auditing of system use and asset access.

Most enterprises need a scalable storage solution. Typical implementations bolt the security model and accounting on the side and are subject to hacks along both interfaces (security and accounting). In MeTTaCycle these two have been seamlessly integrated and baked into searchable, transactional storage.

3 High-performance MeTTa Kernel

3.1 Theories of computation

In this section, we attempt to implement a couple of theories of computation in MeTTa and uncover some missing features and some problems with the current implementation.

3.1.1 The theory of lambda calculus

MeTTa can model the operational semantics of certain virtual machines using a pattern similar to those of platforms like K Framework [6]. Here's a formalization of the lambda calculus in MeTTa, with an evaluation strategy that reduces terms everywhere except under a lambda:

```
; -- The theory of lambda calculus

(: T Sort)
(: App (-> T T T))
(: Lam (-> (-> T T) T))
(= (App (Lam $f) $v) ($f $v))   ; Beta reduction

; -- Named functions due to lack of lambda in MeTTa

(: ident (-> T T))
(= (ident $x) $x)

(: omega (-> T T))
(= (omega $x) (App $x $x))

(: Omega (-> T T))
```

```
(= (Omega $x) (App (Lam omega) (Lam omega)))

; -- Example reductions

!(App X (App (Lam ident) Y))    ; [(App X Y)]
!(Lam Omega)                    ; [(Lam Omega)]
;!(App (Lam omega) (Lam omega)) ; infinite loop
```

The theory introduces `T` as a sort, essentially the generator in the grammar for terms. It then declares the term constructors: `App` takes two terms and produces a term, while `Lam` takes a function from terms to terms and produces a term. By using a function type, the theory can avoid modeling explicit substitution and alpha equivalence as part of the grammar by bootstrapping `Lam` using the binders in MeTTa itself. Finally, the theory declares the beta rule.

Note that there's nothing in the theory corresponding to the rule defining the evaluation strategy:

$$\frac{\texttt{T} \rightsquigarrow \texttt{T}' \quad \texttt{U} \rightsquigarrow \texttt{U}'}{\texttt{(App T U)} \rightsquigarrow \texttt{(App T' U')}}.$$

The evaluation strategy is hardcoded into the MeTTa interpreter. We could not, for instance, choose to reduce solely in head position:

$$\frac{\texttt{T} \rightsquigarrow \texttt{T}'}{\texttt{(App T U)} \rightsquigarrow \texttt{(App T' U)}}$$

or to reduce under a lambda without considerably more machinery.

3.1.2 The theory of RHO calculus

Unfortunately, MeTTa is also unable to model the RHO calculus [8] (a Reflective Higher-Order pi calculus) in a straightforward way:

```
(: P Sort)
(: N Sort)
(: Zero P)
(: Par (-> P P P))
(: Send (-> N P P))
(: Recv (-> N (-> N P) P))
(: At (-> P N))
(: Run (-> N P))
(= (Par (Send $chan $proc) (Recv $chan $cont)) ($cont (At $proc)))
(= (At (Run $n)) $n)
(= (Run (At $p)) $p)

; -- Problem: no support for structural equivalence.
; -- RHO terms form a commutative monoid, but in MeTTa
; -- the commutativity rewrite causes an infinite loop.
```

```
; (= (Par $p1 $p2) (Par $p2 $p1)) ; commutative
(= (Par $p Zero) $p) ; unital
(= (Par ((Par $p1 $p2) $p3) (Par $p1 (Par $p2 $p3)))) ; assoc.

; -- Helpers

(: nil (-> N P))
(= (nil $n) Zero)
(: chan (-> N))
(= (chan) (At Zero))

; -- Problem: reduction under a Send.
; -- According to RHO semantics, the printed process below should
; -- not reduce, but MeTTa reduces it to (Send (At Zero) Zero).

(: Proc (-> P))
(= (Proc) (Par (Send (chan) Zero) (Recv (chan) nil)))
!(Send (chan) (Proc))
```

MeTTa does well at modeling the sorts and term constructors. There are sorts for Processes and Names of channels, five term constructors, and three rewrite rules. But RHO calculus terms (as in all pi calculi) should be considered only up to structural congruence. The term constructors form a commutative monoid under `Par`, where `Zero` is the unit. MeTTa has no concept of structural congruence.

One could argue that detecting whether two words are in the same equivalence class is, in general, undecidable, and any real implementation would use something like a normal form or a hash table to eliminate the detection problem. The Rholang interpreter, in fact, uses such a strategy. But the specific implementation details should not be part of the abstract description of the language.

RHO calculus—like all pi calculi—is a model of concurrent processes, and therefore can have races. When multiple `Send`s or `Recv`s are competing on the same channel, RHO calculus makes a single nondeterministic choice of the winner of the race. MeTTa takes all paths, so even very simple RHO calculus programs require exponential space and time to execute on the MeTTa interpreter.

Finally, RHO calculus also forbids reduction under a Send. For example, the term

`(Send (chan) Proc)`,

where

`Proc = (Par (Send (chan) Zero) (Recv (chan) nil))`

should not reduce. The process `Proc` will always reduce to `Zero` if it is permitted to do so; but under the RHO calculus semantics, `Proc` should be suspended until

it is received and executed with the `Run` constructor. This matters because when `Proc` is received, it could be executed in the context of another `Recv` that could interfere on `(chan)` and the two `Recv`s would race to claim the `Send`.

3.1.3 Other theories

Other features missing from MeTTa that impair its ability to implement theories of computation include:

- The lack of a `lambda` grounded atom for defining functions inline.
- No way to force evaluation of a subexpression.
- No way to add facts to the database as the result of some computation.
- A single shared fact database.
- No way to have theories execute concurrently.
- No transactions.
- No proof of soundness of a type system.
- No automatic way to update a type system to include two theories of computation made to interact.

3.2 A way forward

The mathematician William Lawvere's research underpins all approaches to formalizing theories of computation, so we review his contributions and more recent generalizations below. In this section, we show how to use these theories to generate a reduction graph for a theory of computation. The vertices of the graph are the possible states the computation can be in, and the edges are the possible state transitions. The section culminates in the notion of an interactive graph-structured lambda theory (GSLT), which provides a language for expressing computational theories that includes the ability to express reduction strategies.

From any of these theories, we can derive a sound type system. Because it is derived rather than invented, it allows an intelligence to have type systems for interacting theories of computation and to rederive them when it develops a new theory for some domain.

3.2.1 Lawvere's algebraic theories

In his 1963 PhD thesis [7], William Lawvere introduced the concept of an "algebraic theory". His interest was in constructing categories of algebraic structures like monoids, groups, rings, fields, and so on. A presentation of an algebraic theory looks very much like the MeTTa presentation of a theory of computation.

A presentation of an algebraic theory consists of:

- A *sort*, say T.
- A set of *function symbols* f_i, each with a finite *arity*. If f_i has arity $n > 1$, we write $f_i : T^n \to T$, where $T = T^1$ and $1 = T^0$.
- A set of *equations* between terms generated by the function symbols and a set of free variables.

For example, here's a presentation of the algebraic theory of groups Th(Grp):

- A sort G.
- A function symbol $m\colon G^2 \to G$ for the multiplication.
- A function symbol $e\colon 1 \to G$ for the identity.
- A function symbol $i\colon G \to G$ for the inverse.
- An equation $m(m(a,b),c) = m(a,m(b,c))$ for the associative law.
- Equations $m(a,e) = a$ and $m(e,a) = a$ for the unit laws.
- Equations $m(a,i(a)) = e$ and $m(i(a),a) = e$ for the inversion laws.

The theory itself is essentially[1] the free category with finite products on the data above.

A model M of the theory is a product-preserving functor from the the theory to the category Set. It picks out:

- a set $M(G)$ of elements of the group
- a multiplication function $M(m)\colon M(G)^2 \to M(G)$
- a nullary function $M(e) : 1 \to M(G)$ that returns the identity element
- an inversion function $M(i) : M(G) \to M(G)$

such that multiplication is associative, unital, and invertible.

Lawvere proved that when an algebraic gadget is a set equipped with some functions satisfying some equations, the category of gadgets and gadget homomorphisms is equivalent to the category of product-preserving functors from the theory of gadgets to Set and the natural transformations between them. He also proved that the theory is the opposite of the category of finitely generated gadgets.

[1] There's a little more to the definition of a theory, but it's not critical to this discussion.

3.2.2 Multi-sorted and graph-structured theories

Trimble [10] generalized Lawvere theories, which have a single sort, to multi-sorted theories. This lets us add interactions between different types of gadget; for example, a group action has both a group and a set for it to act on. But more importantly for this discussion, it adds the ability to talk about edges in a reduction graph.

We can add a sort (say, R for rewrites or reductions), function symbols s and t for the source and target of the rewrite, and function symbols to generate the grammar of the reductions. With this approach, we can present the theory of SKI combinators (one of the earliest theories of computation, from 1924) with the reduction strategy that we only reduce in the head of an application:

1. A sort T for terms.
2. A sort R for rewrites.
3. Term function symbols $s, t\colon R \to T$ for the source and target of a rewrite.
4. A term function symbol $(- -)\colon T^2 \to T$ for application.
5. Term function symbols $S, K, I\colon 1 \to T$ for the combinators.
6. A rewrite function symbol $\sigma\colon T^3 \to R$ for reducing S in the head.
7. A rewrite function symbol $\kappa\colon T^2 \to R$ for reducing K in the head.
8. A rewrite function symbol $\iota\colon T \to R$ for reducing I in the head.
9. A rewrite function symbol $\eta\colon R \times T \to R$ for nesting reductions in the head.
10. Equations $s(\sigma(x, y, z)) = (((S\ x)\ y)\ z)$ and $t(\sigma(x, y, z)) = ((x\ z)\ (y\ z))$.
11. Equations $s(\kappa(x, y)) = ((K\ x)\ y)$ and $t(\kappa(x, y)) = x$.
12. Equations $s(\iota(x)) = (I\ x)$ and $t(\iota(x)) = x$.
13. Equations $s(\eta(r, x)) = (s(r); x)$ and $t(\eta(x)) = (t(r)\ x)$.

Note that we have function symbols for generating two grammars, the term grammar (items 1, 3-5) and the rewrite grammar (items 2, 6-9). The equations say what the source and target of each rewrite is. Items 9 and 13 encode the reduction strategy of reducing only in the head of an application:

$$\frac{t \overset{r}{\rightsquigarrow} t'}{(t\ x) \overset{\eta(r,x)}{\rightsquigarrow} (t'\ x)}.$$

We'll adopt syntactic sugar for specifying the source and target of a rewrite and declare that $\rho(\vec{x})\colon T(\vec{x}) \rightsquigarrow U(\vec{x})$ is sugar for the pair of equations $s(\rho(\vec{x})) = T(\vec{x})$ and $t(\rho(\vec{x})) = U(\vec{x})$.

3.2.3 Graph-structured lambda theories

The inputs to function symbols in algebraic theories can only have product types. Lambda theories expand the possible inputs to have function types as well. With graph-structured lambda theories (GSLTs), we can present the theory of lambda calculus with the reduction strategy that we only reduce in the head of an application:

1. A sort T for terms.
2. A sort R for rewrites.
3. Term function symbols $s, t \colon R \to T$ for the source and target of a rewrite.
4. A term function symbol $(- \ -) \colon T^2 \to T$ for application.
5. A term function symbol $[-] \colon (T \to T) \to T$ for abstraction.
6. A rewrite function symbol $\beta \colon (T \to T) \times T \to R$ for beta reduction.
7. A rewrite function symbol $\eta \colon R \times T \to R$ for nesting reductions in the head.
8. Equations $\beta(f, v) \colon ([f] \ v) \rightsquigarrow f(v)$.
9. Equations $\eta(r, x) \colon (s(r) \ x) \rightsquigarrow (t(r) \ x)$.

Items 5 and 8, like the MeTTa theory of lambda calculus above, use the built-in binders of the lambda theory to avoid the boilerplate of explicit substitution and alpha equivalence. Items 7 and 9 encode the reduction strategy in the same way as the SKI calculus above.

We can also present the theory of RHO calculus as a GSLT:

1. Sorts P and N for processes and names, respectively.
2. A sort R for rewrites.
3. A process function symbol $0 \colon 1 \to P$ for `Zero`.
4. A process function symbol $| \colon P^2 \to P$ for `Par`.
5. A process function symbol $! \colon N \times P \to P$ for `Send`.
6. A process function symbol $? \colon N \times (N \to P) \to P$ for `Recv`.
7. A name function symbol $@ \colon P \to N$ for `At`.
8. A process function symbol $* \colon N \to P$ for `Run`.
9. A rewrite function symbol $\chi \colon N \times P \times (N \to P) \to R$ for the communication event.
10. A rewrite function symbol $\pi \colon R \times R \to R$ for parallel execution.

11. Equations $\chi(x, p, q)\colon x!(p) \mid x?(q) \rightsquigarrow q(p)$ for a send and a receive synchronizing on the name x.

12. Equations $\pi_1(r_1, x)\colon s(r_1) \mid x \rightsquigarrow t(r_1) \mid x$ for reductions in a par context.

13. Equations $\pi_2(r_1, r_2)\colon s(r_1) \mid s(r_2) \rightsquigarrow t(r_1) \mid t(r_2)$ for parallel reductions.

14. An equation $p \mid q = q \mid p$ for commutativity of `Par`.

15. An equation $(p \mid q) \mid r = p \mid (q \mid r)$ for associativity of `Par`.

16. An equation $0 \mid p = p$ for `Zero` being the unit.

17. Equations $*@p = p$ and $@ * n = n$ for suspension and execution.

Under these rules, reduction only happens (as it should) in the context of a `Par` at the top level (items 9-13), not in a send or receive.

3.2.4 Interactive GSLTs

In the SKI calculus and lambda calculi, reduction is triggered by combining a function and a value in an application context. In pi calculi and the ambient calculus [3], reduction is triggered by combining a send and a receive in par context. In a Turing machine, reduction is triggered by being in a tape context (as opposed to a halting context). In cellular automata, reduction is triggered by being in a neighborhood context (which is always the case). Every theory of computation of which we're aware has a term context that triggers reduction. We can model reduction contexts by picking a binary function symbol to be the trigger for reduction; we call this symbol the *interaction* and denote it formally by $\odot$; for example, in pi calculus, $\odot = |$.

Modal logic is a kind of logic used to reason about possibility and necessity. In the context of theories of computation, we're interested in identifying properties of terms, like, "Does this term necessarily terminate?" or "Is it possible for a client of this smart contract to extract all the tokens?" Once we've chosen the interaction, we can derive certain useful modal types.

For example, consider the modal type $\langle(-\ A)\rangle B$ of lambda terms that when applied to a value of type A possibly reduce to a value of type B in one step. This modal type corresponds precisely to the arrow type $A \to B$. There is nothing explicit about the arrow type in the theory of the lambda calculus; it simply falls out of the theory by considering terms in the first slot of the interaction.

Similarly, consider the modal type $[-\mid A\rangle B$ of RHO processes that when juxtaposed with a process of type A necessarily reduce to a process of type B in one step. This modal type corresponds precisely to Caires' rely-guarantee modality $A \triangleright B$ [2].

These modal types will play an important role in the generated type system described in the next section.

3.3 Generating an interpreter and a type system

In this section we consider the problem of generating an efficient interpreter and a type system from a GSLT.

3.3.1 Fine- and coarse-grained GSLTs

In the RHO calculus, the terms form a commutative monoid. However, there are $n!$ permutations of a term with $n-1$ parallel atomic processes; if we had to enumerate all the permutations until we found a term in the right form to reduce, it would be absurdly slow. Even worse, the general problem of determining whether two words are in the same equivalence class is undecidable. Actual implementations of process calculi will do things like replace variables with keys into a hash table, store all the receiving processes in buckets under their key, and then iterate over the sends looking for matches. Therefore, when generating a vm and a type system from a GSLT, a language implementer will likely want to provide two GSLT presentations and a proof.

The first presentation will be of the fine-grained GSLT with all the optimizations and implementation details; it will only have equations pertaining to the source and target of rewrites and will have logic for things like reducing to a normal form or using a hash table. We can derive a type system from such a GSLT, but it will likely be too complicated to be used by developers writing programs in the language.

The second will be a coarse-grained GSLT that abstracts away the implementation details and things like specific choices of normal form. This is the GSLT that we feed to the algorithms above to generate a useful type system. From an arbitrary GSLT we can automatically derive a "native" type system [11]—the internal language of the topos of presheaves on the theory—and from an interactive GSLT, we can derive a smaller but still very useful dependent type system similar to Barendregt's lambda cube [5]. The modalities in the previous section play the role analogous to that of the dependent product in the lambda cube.

The proof should show that the fine-grained GSLT is a valid implementation of the coarse-grained one. One way to do this is with a necessity-preserving functor between the free quivers on the reduction graphs. Possibility is preserved by any functor between the quivers: given a possible rewrite from one term to another in the coarse-grained GSLT, there is a path of possible rewrites between the corresponding terms in the fine-grained GSLT. Preserving necessity means that given a necessary rewrite from one term to another in the coarse-grained GSLT, there is a path of necessary rewrites between the corresponding terms in the fine-grained GSLT.

3.3.2 Compilation to Rholang

Rholang is an extension of the RHO calculus that seamlessly combines synchronous datatypes with the best-studied approach to concurrency, the π-calculus, to provide an efficient, transactional knowledge store. Given a finitely-presented

fine-grained GSLT, we would like to produce a Rholang program that implements it. The general strategy is to send the state of the system on a channel, then produce a sum of "edge" processes that implement rewrites out of the state. Each edge process in the sum attempts to consume the current state and produce the edge's target state.

Enumerating rewrites We can't produce a sum of all the edge processes out of a state simultaneously due to a finite amount of memory. In an implementation of the lambda calculus with a reduction strategy that allows beta reduction anywhere within a term, it is easy to produce an exponential number of possible rewrites out of a term. For example, suppose we have a bunch of applications of the identity combinator `I =` $\lambda x.x$ to itself:

```
((I I) ((I I) ((I I) ((I I) ((I I) I)))))
```

There are five different places in this lambda term where a beta reduction could occur, so there are $2^5 = 32$ different parallel reductions that could occur.

Even worse, consider the following modification of RHO calculus' Comm rule:

$$\text{Comm}(x, K, Q_1, Q_2) : \text{Send}(x, Q_1) \mid \text{For}(x, K) \rightsquigarrow Q_2.$$

This says that any interacting pair of processes can evolve to *any other term*! It is useless computationally, but demonstrates that in principle we can have an infinite number of rewrites out of a source term.

We can write an interpreter with a process that unfolds the sum dynamically. At each step, the interpreter may choose to synchronize with of the existing edge processes in the sum or to recurse and generate another edge process.

```
Intepreter = EdgeGen(state, 0)

EdgeGen(state, n) =
    EdgeProcess(state, n) + EdgeGen(state, n + 1)
```

This technique produces a single-threaded interpreter that includes in its enumeration parallel rewrites on parts of the state. It suffices for an interpreter, but does not make good use of the massive parallelization that RSpace provides.

Adding parallelism A GSLT has a finite number of top-level and in-context rewrite constructors. Top-level rewrites unify against a whole term, while in-context rewrites unify against part of a term and then recurse. The recursion must terminate because the terms are of finite length.

However, the interpreter is not given rewrites, it's given a term that may be the source of a rewrite. Matching the state agains the source of a rewrite constructor is a unification problem. The implementation above needs to verify that the entire term is the source of a rewrite in order to add that edge process

to the sum. Rather than preverifiy and sum the resulting processes, we can delegate the verification to processes that are *potentially* edges out of the state and simply run them in parallel. Any potential edge process that manages to verify that the current state really is the source of that edge can then send a message to a channel waiting for a winner.

RHO calculus only has public names, so if we tried this technique in RHO calculus, it would leave a lot of garbage laying around:

```
@Nil!(1) | @Nil!(2) | for (winner <- @Nil) stdout!(*winner)

    | 2 wins
    V

@Nil!(1) | stdout!(2)
```

However, Rholang also allows private names, like those used by pi calculus. If we use private names, the loser can get garbage collected because we're guaranteed that no other process can synchronize with it:

```
new x in { x!(1) | x!(2) | for (winner <- x) stdout!(*winner) }

    | 2 wins
    V

new x in { x!(1) | stdout!(2) }

    || x not free in stdout!(2)

new x in { x!(1) } | stdout!(2)

    || no synchronization possible

Nil | stdout!(2)

    || monoid laws

stdout!(2)
```

This design pattern allows a much simpler interpreter that has one potential edge process for each rewrite constructor in the GSLT. If an in-context rewrite constructor matches the state, the interpreter forks new processes that independently try to verify that the substructure is the source of a rewrite, then joins those processes and signals that it found an actual edge.

Detecting optimization opportunities There are often more opportunities for optimization, but they only apply in certian circumstances. Detecting opportunities for optimization is a standard part of interpreter design, and will be

important for a compilation pipeline targeting Rholang. For example, in a programming language with functions and an eager reduction strategy, it doesn't make sense to decompose the state from the top level at each step; instead, subexpressions should be reduced completely before moving up in the syntax tree. Standard heuristics can be brought to bear on the sets of rewrite constructors, and automated techniques exist for generalizing specific instances of optimizations to the most general applicable situation [9].

3.4 RSpace

The Rholang interpreter uses a very efficient data structure called **RSpace** for storing continuations and matching sends with receives. To a first approximation, the Rholang interpreter is just a parser sitting on top of RSpace.

In the simplest cases, the names on which processes are sending and receiving get hashed, and if there's a process of the opposite polarity waiting in the hash table, they can synchronize. RSpace also allows processes to bind variables using patterns. This lets programmers dispatch messages on the same channel to different processes based on the content of the messages, and is the basis of the design of the interpreters above.

3.4.1 RSpace and MORK

MORK (MeTTa Optimal Reduction Kernel) is a powerful functional language for manipulating graph databases in memory. It represents sets of paths using a trie, which can be exponentially smaller than the original dataset. It can also act on all the paths in a subtrie simultaneously, which can be exponentially faster than mapping over a data set. MORK was designed with provable efficiency in mind: each operation provides complexity guarantees. Using MORK lets a programmer make accurate predictions about the computational complexity of a query.

However, MORK does not have any transactional semantics or primitives for concurrency. As such, RSpace and MORK are complementary languages, well-suited to each other. Rholang 1.2 will embed MORK as a datatype in the same way that it embeds numbers, strings, lists, maps, and sets. It will also use MORK for unification with patterns in receiving processes instead of merely binding a single variable.

3.5 Queries

There is a long tradition of associating types with predicates. One of the earliest kind of query was whether a term halts or not. This is, as Church and Turing showed, undecidable. But the subsequent development of type systems for the lambda calculus focused on that question, and any well-typed program in any of the type systems of the lambda cube is guaranteed to halt.

Many modern languages also use predicates for types. TypeScript, for example, includes a notion of narrowing the type of a value within `if` blocks whose

condition tests some property of the value. The native type theory mentioned above is the internal language of the topos of presheaves on the theory, and this internal language includes the ability to express predicates; but more generally, we can simply ask for witnesses of any particular type in a type system.

Finding such witnesses is the subject of an enormous literature and is outside the scope of this paper, but restricted problems can be solved quickly (e.g. MORK handles structural type queries very quickly). Certainly Prolog's backtracking approach is a brute-force way of finding witnesses, but there are many other ways, and an interpreter supporting queries should expose many different methods with clear complexity guarantees.

The automatically generated type systems above give us grammars in which to express queries in the form of an expanded GSLT including both the original operational semantics and new type-constructing function symbols. Since queries are about terms in a GSLT and the type system itelf is expressed as a GSLT, queries can produce new queries in which facts are added, subtracted, or transformed.

The notion of a theory of computation is very general; one could, for example, model a cell as a computing device where terms are cell states and rewrites involve molecules and cell signaling pathways. Healthy and diseased cells become types expressible in the type system, and we have modal types corresponding to small molecules that in the context of a diseased cell lead to a healthy cell.

4 Decentralizing MeTTa

4.1 Tokenization and capabilities

5 Tokenized security

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

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

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

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

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

$$\text{COMM-COSIGNED-PAR-EXTERNAL-SEQUENTIAL}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\{\mathsf{for}(t \leftrightarrow x)P\}_{s_1} \mid \{\mathsf{for}(u \leftrightarrow x)P\}_{s_2} \mid s_1 \& s_2 : T \to \{P\dot{\sigma} | Q\dot{\sigma}\}_{s_1 \& s_2} \mid T}$$

$$\text{COMM-COSIGNED-PAR-EXTERNAL-CONCURRENT}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\{\mathsf{for}(t \leftrightarrow x)P\}_{s_1} \mid \{\mathsf{for}(u \leftrightarrow x)P\}_{s_2} \mid s_1 : T_1 \mid s_2 : T_1 \to \{P\dot{\sigma} | Q\dot{\sigma}\}_{s_1 \& s_2} \mid T_1 \mid T_2}$$

$$\text{COMM-SIGNED}\quad \frac{P \to P'}{\{P\}_s \mid s : T \to \{P'\}_s \mid T}$$

$$\text{COMM-COSIGNED-PAR-INTERNAL}\quad \frac{P \to P'}{\{P\}_{s_1 \& s_2} \mid s_1 : T_1 \mid s_2 : T_2 \to \{P'\}_s \mid T_1 \mid T_2}$$

5.1 Embedding the kernel in a consensus mechanism

5.1.1 Correct-by-construction Casper

TBD

5.1.2 TODA IP

TBD

6 Deployment and integration

6.1 Integration with the OS

6.2 Integration with social media

7 Roadmap

8 Conclusion

Our proposed architecture addresses the defects of the current MeTTa implementation:

- GSLTs can specify the semantics of any discrete theory of computation
- From a GSLT, we can generate an efficient interpreter for that theory
- From a GSLT, we can generate a sound type system customized to that theory

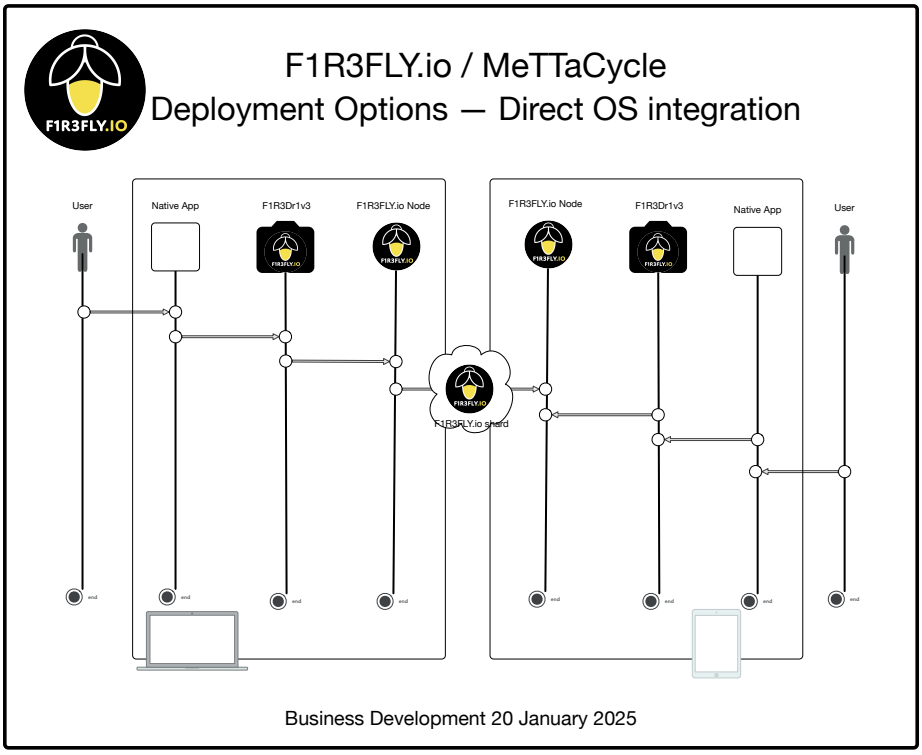

Figure 1: OS/File system deployment

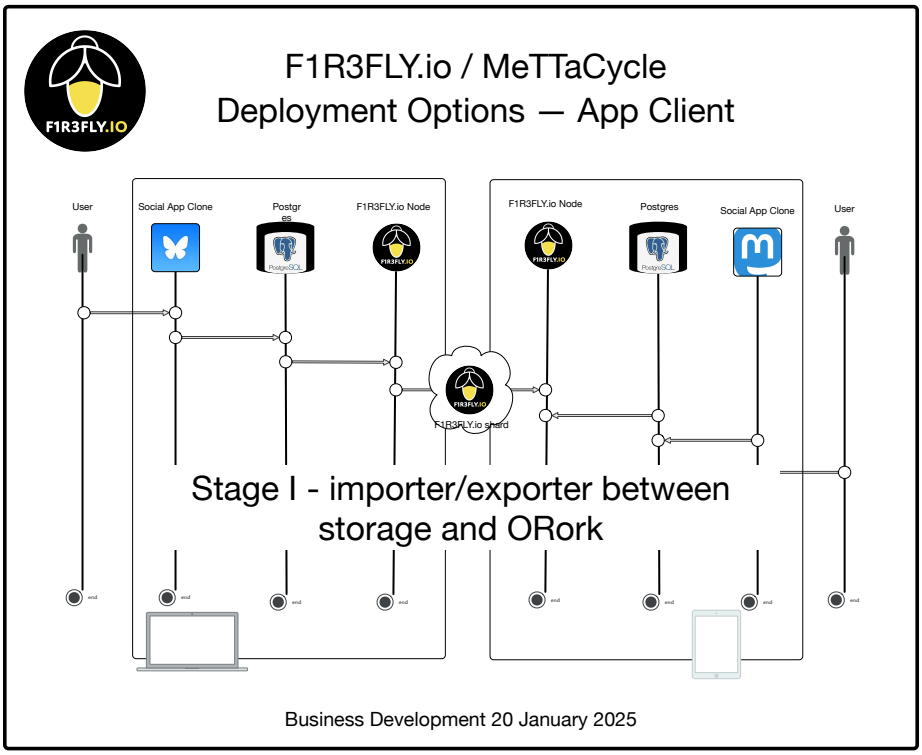

Figure 2: Social deployment

- GSLTs allow defining functions inline.
- GSLTs let the programmer specify exactly when subexpressions can reduce.
- Rather than have a single shared fact database, the facts become part of a query, and the intepreter can produce new fact databases in in response to queries.
- Theories can be combined in multiple ways, including concurrent execution.
- State updates are transactional.
- The automatically generated type systems are provably sound.
- When two theories are combined, the system can automatically generate a new type system encompassing both and their interactions.

References

[1] Barendregt, Henk (1991). "Introduction to generalized type systems." *Journal of Functional Programming*, 1(2): 125–154.

[2] Caires, Luis (2007). "Logical Semantics of Types for Concurrency." In *Algebra and Coalgebra in Computer Science*, Second International Conference, CALCO 2007, Bergen, Norway, August 20–24, 2007, Proceedings. `http://ctp.di.fct.unl.pt/~lcaires/papers/CALCO-Caires-1.0.pdf`

[3] Cardelli, Luca, and Andrew D. Gordon (2000). "Mobile ambients." *Theoretical Computer Science*, 240: 177–213.

[4] Goertzel, Ben, and L. Gregory Meredith. Private communication.

[5] [Placeholder for the Hypercube functor paper reference; "cite the hypercube functor paper"].

[6] K Framework. `http://kframework.org`.

[7] Lawvere, William (1963). "Functorial Semantics of Algebraic Theories" (PhD Thesis). `http://www.tac.mta.ca/tac/reprints/articles/5/tr5abs.html`.

[8] Meredith, L. Gregory, and Matthias Radestock (2005). "A Reflective Higher-order Calculus." *Electronic Notes in Theoretical Computer Science* 141(5), 49–67, 22 December 2005. `https://doi.org/10.1016/j.entcs.2005.05.016`

[9] Tate, Ross, Michael Stepp, and Sorin Lerner (2010). "Generating Compiler Optimization from Proofs." In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 389–402, Madrid, Spain. ACM, New York, NY, USA. `http://www.cs.cornell.edu/~ross/publications/proofgen/`.

[10] Trimble, Todd (2018). "Multisorted Lawvere Theories," on the nLab. `https://ncatlab.org/toddtrimble/published/multisorted+Lawvere+theories`.

[11] Williams, Christian, and Michael Stay (2021). "Native Type Theory." *Proceedings of the Fourth Annual Conference on Applied Category Theory.* EPTCS 372 (2022), 116–132.