MeTTaCycle Architecture Proposal

Contents

1 Introduction

MeTTaCycle is the codename for a new decentralized AI platform. The network will have agents, both human and AI, who can share digital assets of all kinds, attenuate and delegate access to those assets, and make powerful queries about each other's content.

We anticipate at least two interface points between humans and the network; both of these will be based on existing user interfaces with which users are already familiar. The first is through the operating system, where F1R3Dr1v3 will be mounted into the filesystem. The second will be based on existing open source social networking software, currently BlueSky and Mastodon.

MeTTaCycle, as its name suggests, will feature a version of the MeTTa programming language. The driving concept for MeTTa is the idea that intelligence opportunistically creates theories of computation to suit the domain in which the intelligence is operating. MeTTaCycle intends to connect many theories operating in many domains, and to enable reasoning about these interactions.

1.1 The MeTTa Kernel

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, the alpha version of MeTTa has limitations with respect to the ultimate goals for the language. Currently, the alpha version 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 the next version of 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 the alpha version of MeTTa to illustrate some of its limitations. Section 3.2 describes graph-structured lambda theories (GSLTs), which lifts these expressability limitations. Section 3.4 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 3.6 returns to the need to query a fact database and examines the kinds of interesting query one can express in this system.

1.2 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 amplified by 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

2.1 Security versus scaling and throughput

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". Billions of dollars 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 and accounting models 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.

2.2 Data and agents on chain

The real benefit to AI-oriented applications is not merely security. Consider, the current generation of LLMs were trained primarily on data stolen from the Internet. The next generation of models will not be. Instead they will be trained on data hard won and costing the companies who developed it billions of dollars. These companies will not just hand over their data to OpenAI or Anthropic. Companies like Merck or Pfizer or Astrazeneca are also loathe to spend additional billions developing in-house models.

A decentralized digital asset management system capable of hosting data and agents on chain solves these problems. It allows companies who hold highly valuable data sets to tokenize and monetize their data and companies, like Grok who have developed models as a service, to bring their services to the chain. We view this as the next high growth marketplace. Companies like OpaqueSystems share our view, but not our technology or approach.

2.3 Integration into the technological ecosystem

In order for this approach to work MeTTaCycle needs seamless integration into the technological ecosystem. For example, MeTTaCycle is already mountable as a file system on all major desktop platforms. The chain just looks like another folder. Users can use their existing file browsers to interact with the chain, that is simply drag and drop audio, video, and text files directly onto chain and from chain. From their point of view MeTTaCycle looks like a fancy DropBox or GoogleDrive.

Beyond this, a dominant mode of interaction between users is social media. People have been trained on Twitter, Facebook, and the like for more than a decade. MeTTaCycle recognizes this fact of user behavior in the market place. Already the platform is demonstrating integration with BlueSky so that MeTTaCycle acts as a backbone between independent BlueSky instances.

The plan is to integrate wallet and agent features into a fork of BlueSky and Mastodon shortly after. This opens up a world of new user opportunities. Tokenization takes on entirely new meaning in this setting with users being able to share access to data sets, compute, and agents directly with each other as part of social communication. Likewise, they can moderate and control access to feed-delivered content via tokens. Finally, they can create, launch, interact with, shut down, and *share* agents directly with each other.

2.3.1 Interoperability with other networks

With the proliferation of the BFTNs the question of interoperability looms large, both in terms of interoperation of different instances of the same network, and interoperation of different kinds of networks. In the first case private instances of a blockchain may be operating in different divisions of a single company and it becomes useful or necessary to have these instances interoperate to serve use cases that cross these divisions. A similar sort of scenario arises when rolling out different versions of a single network. In the second case user network interactions may span multiple networks, travel being a prime example. One can easily imagine the airlines settle on one network, while the hospitality providers settle on another. The user doesn't actually care. They just want to book a trip including airfare and hospitality.

On the one hand MeTTaCycle is designed to be compositional. Shards can be stitched togher into larger networks. On the other hand, the channel-based model of MeTTaCycle makes it possible to treat other networks as if they were shards. Still MeTTaCycle has larger aims. As will be discussed, the technology offered here makes it possible for MeTTaCycle nodes to participate as first class citizens in other networks. This opens up a new world of possibilities for how to bring features, like data on chain or agents on chain to other networks.

3 High-performance MeTTa Kernel

3.1 Theories of computation

In this section, we attempt to implement a couple of theories of computation in the alpha version of MeTTa and uncover some missing features and limitations 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:

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

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

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

or to reduce under a lambda without considerably more machinery.

3.1.2 The theory of RHO calculus

Unfortunately, the alpha version of 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 research version of 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 the alpha version of 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.

3.2.2 Multi-sorted and graph-structured theories

Trimble [11] 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.

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

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 $[- \mid A]B$ 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 A GSLT for MeTTa

In the same way that there is a grammar for BNFC, there is a GSLT $\mathsf{Th}(GSLT)$ for presenting finitely presentable GSLTs. It has no rewrites, because it's just describing the grammar of GSLTs. We can extend $\mathsf{Th}(GSLT)$ with a grammar for facts and queries, with rewrites for processing those queries; see section 3.6 below for more details.

3.3.1 MeTTaIL

MeTTaIL takes the idea presented above of a self-describing GSLT for describing GSLTs seriously, and provides a clean intermediate language for representing theories in a way that is more compiler-friendly than the alpha version of MeTTa's design. That is, MeTTaIL files represent compilation units in much the same way that a file in Java represents a compilation unit.

Beyond this, MeTTaIL is the input language for the algorithm for generating an interpreter and the algorithm for generating a type system. The GitHub repo for the self-describing spec for MeTTaIL is linked *here*. To make this paper a little more self-contained we have included an appendix with a summary of the spec: 11. The repo contains as a test an example of a MeTTaIL specification for rholang which is linked *here*. Again, to help with some of the context switching, we have included this content as an appendix 14.

The rholang example is of particular note in the decentralization space. Rholang is a smart contracting language. We fully anticipate being able to provide MeTTaIL specs for other smart contracting languages including Solidity, as well as Solana and Cardano's smart contracting languages. Taken together with the fact that MeTTaCycle nodes are capable of operating in heterogeneous consensus environments, in this way we aim to allow MeTTaCycle nodes to be fully compliant nodes in other chains. However, these nodes will also allow data on chain and agentic computation.

3.4 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.

3.4.1 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 [12]—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.

3.4.2 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:

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

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 [10].

3.5 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.

3.5.1 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.

3.6 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

Rholang is an object capapbilities platform. On a single logical machine like a blockchain, private names created with the `new` operator are unforgeable[2] bearer tokens for access to a process. On an open network, the only things that can be transmitted are bits, so names are necessarily forgeable. But they can be made unguessable, like the long random strings in API keys.

[2]A capability is forgeable if you can create an instance of it given only bits. Object references in memory safe languages are unforgeable, since even if you have the memory address of an object, there's no way to turn it into an object reference.

These names can be attenuated and delegated, and interposing processes can enable arbitrarily fine-grained security patterns [9]. For example, Alice can make access to a process P listening for messages on channel p revocable by interposing another forwarder process F that listens on two channels f and r. Alice shares revocable access to P by sending Bob the name f, but she keeps r to herself. F forwards to P on p any messages that it receives on f from Bob until F receives a revocation trigger from Alice on r. At that point, F shuts down and Bob has no way to contact P.

MeTTaCycle, however, will also provide metered access through the use of tokens.

4.1 Tokenization and capabilities

This tokenized security model is *generated* from the operational semantics. Here, for example, we apply the method to the operational semantics outlined in the MeTTa-calculus.

$$\begin{array}{ll} \textsc{security-tokens} & \textsc{secured-processes} \\ T ::= () \mid s \mid T : T & S ::= \{P\}_s \mid T \mid S|S \end{array}$$

$$\begin{array}{l} \textsc{multi-party-sigs} \\ s ::= () \mid \mathsf{hash}(<signature>) \mid s\&s \end{array}$$

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

$$\text{COMM-}\textsc{cosigned-par-external-sequential}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\{\mathsf{for}(t \texttt{<->} x)P\}_{s_1} \mid \{x!(u);Q\}_{s_2} \mid s_1\&s_2 : T \rightarrow \{P\sigma|Q\sigma\}_{s_1\&s_2} \mid T}$$

$$\text{COMM-}\textsc{cosigned-par-external-concurrent}\quad \frac{\sigma = \mathsf{unify}(t, u)}{\{\mathsf{for}(t \texttt{<->} x)P\}_{s_1} \mid \{x!(u);Q\}_{s_2} \mid s_1 : T_1 \mid s_2 : T_1 \rightarrow \{P\sigma|Q\sigma\}_{s_1\&s_2} \mid T_1 \mid T_2}$$

$$\text{COMM-}\textsc{signed}\quad \frac{P \rightarrow P'}{\{P\}_s \mid s : T \rightarrow \{P'\}_s \mid T}$$

$$\text{COMM-}\textsc{cosigned-par-internal}\quad \frac{P \rightarrow P'}{\{P\}_{s_1\&s_2} \mid s_1 : T_1 \mid s_2 : T_2 \rightarrow \{P'\}_{s_1\&s_2} \mid T_1 \mid T_2}$$

The last two rules are easy to understand. The third rule says that a computation housed entirely within the membrane of a signature requires a token provided by the agent(s) who has (have) the signing authority of that signature, while the fourth rule says that a computation housed entirely within the membrane of a multi-party signature may use the tokens of each individual signer.

The first two rules are about computation involving synchronization across security boundaries. The first rule says that for a transactional exchange of data where the receiving computation is signed by one party and the sending computation is signed by another, then a multi-sig token containing the signatures of both parties suffices to allow the synchronization. Meanwhile the second rule says that the same situation can progress by taking a token from the receiving and a token from the sending parting. Note that the resulting computation is now signed by both parties and to progress requires tokens from both parties (see rules three and four).

4.1.1 Unification and other ancillary costs

More generally, the method described is an endofunctor on the category of GSLTs. It is not the only such. For instance, it is worth noting that this functor does not take into account the cost of unification, which in some cases can be considerable. A basic approach is to demand that unification return a cost along with the substitution. The cost can be treated as a constant or a multiple of the cost. For example,

$$\text{COMM-}\textsc{cosigned-par-external-concurrent}\quad \frac{(\sigma, n) = \mathsf{unify}(t,u)}{\{\mathsf{for}(t\texttt{<->}x)P\}_{s_1} \mid \{x!(u);Q\}_{s_2} \mid \underbrace{\underline{s_1} : \ldots : \underline{s_1}}_{n} : T_1 \mid \underbrace{\underline{s_2} : \ldots : \underline{s_s}}_{n} : T_1 \to \{P\sigma|Q\sigma\}_{s_1\&s_2} \mid T_1 \mid T_2}$$

These techniques can be used liberally to treat all manner of additional costs incurred in actual calculations on physical hardware.

5 Fault-tolerance: embedding the kernel in a consensus mechanism

One critical aspect of MeTTaCycle is that it makes virtually no commitment to a particular consensus mechanism. Rather, consensus is viewed as a tool for providing fault-tolerance over a community of nodes. As such, MeTTaCycle provides many different consensus mechanisms out of the box and will provide more as time goes on.

Fault-tolerance means different things in different use cases. For example, high-speed trading where the transactions carry very small data payloads can and should be supported by a different algorithm than data-on-chain where the data payloads can be arbitrarily large. Likewise, computing-at-the-edge use cases where nodes are running on mobile devices need yet a different algorithm.

MeTTaCycle plans to support each of these use cases as part of its roadmap.

5.0.1 Correct-by-construction Casper

CBC – Casper was the result of a collaboration amongst Vitalik Buterin, Vlad Zamfir, and Lucius Gregory Meredith. Amongst its many innovations, it uses ideas from the games semantics of linear logic to prevent equivocation. MeTTaCycle has production implementations in both Scala and Rust of CBC – Casper. Its principle contemplated use case is data-on-chain.

5.0.2 Cassanova

Cassanova was developed by Pyrofex with a focus on high-speed trading.

5.0.3 Cordial miners

The cordial miners consensus algorithm was developed by Shapiro, et with a focus on computing-at-the-edge.

6 Deployment and integration

To aid adoption and provide practical deployment MeTTaCycle comes with integration to major throughways in the creation and dissemination of digital assets: the OS and social media.

6.1 Integration with the OS

In the case of integration with the OS MeTTaCycle uses libFUSE to mount a remote shard as if it is a local file system. Effectively, MeTTaCycle's performance is on par with what users normally experience with Dropbox or Google Drive. Using their local file browser, with no modifications, they can drag and drop assets onto a shard. This allows for users to share data with each other over a decentralized network with the security of common BFTNs, such as BTC or ETH. See the F1R3Dr1v3 repo.

6.2 Integration with social media

In the case of integration with social media MeTTaCycle offers two levels of integration. The first of these is importing and exporting between local caches. For example, Zulip, BlueSky, and Mastodon all use Postgres as their local instance storage. MeTTaCycle allows import and export between their Postgres storage and the shard. Thus, MeTTaCycle becomes a bus between different instances – without modification of the social media application stack.

The second of these is a tighter integration involving modifications to the social media stack in order to add features made possible by tokenization of capabilities and the integration of agentic AI.

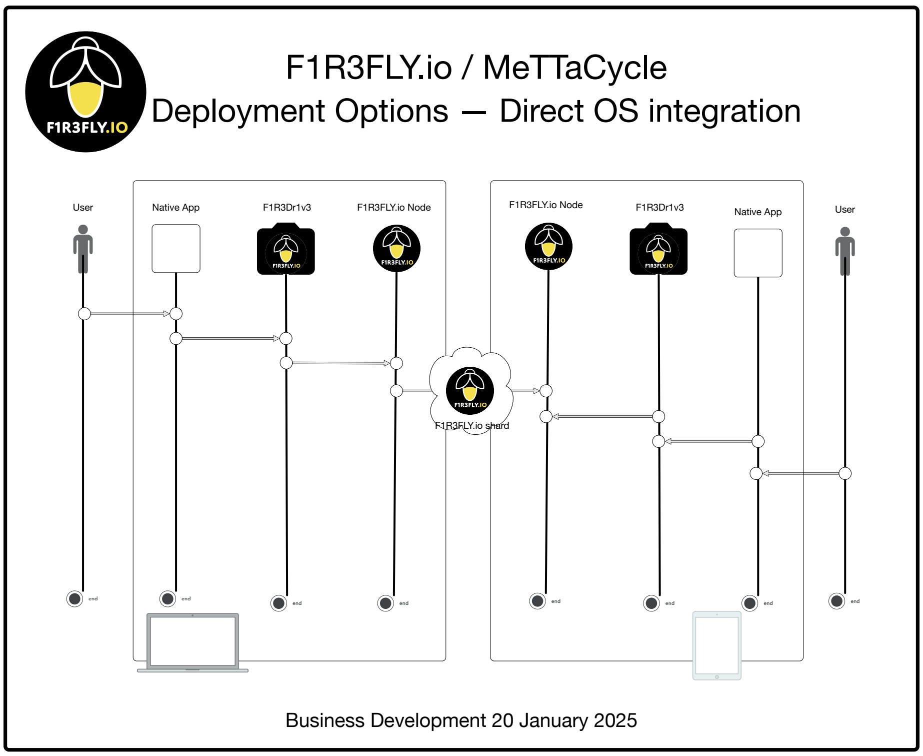

Figure 1: OS/File system deployment

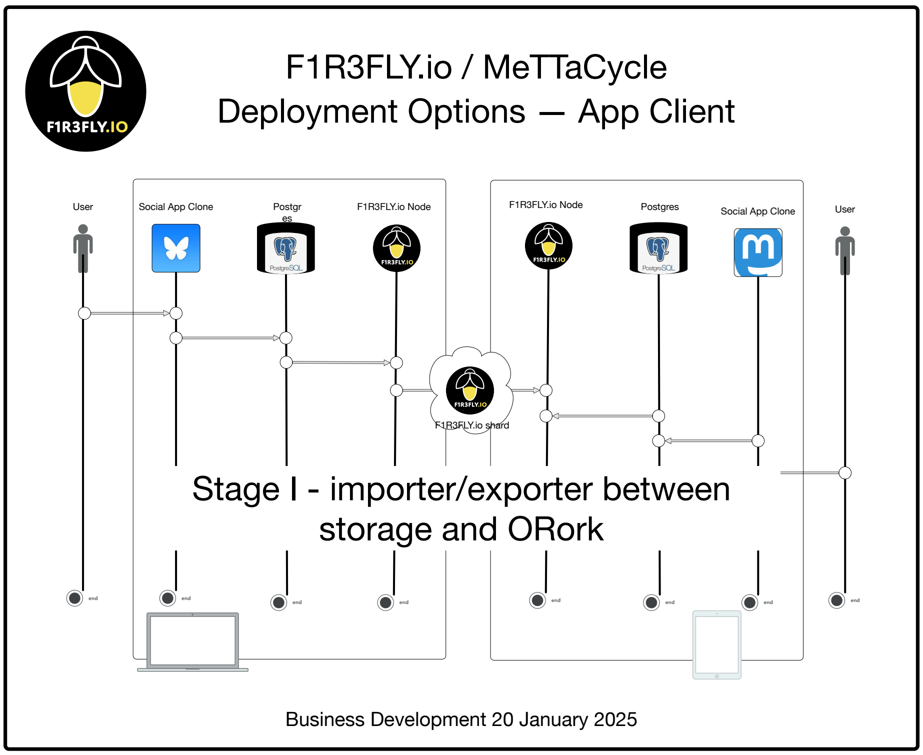

Figure 2: Social deployment

7 Roadmap

Please refer to the live roadmap planning document linked *here*.

8 MeTTaIL and AI

The aim of this section is to suggest that MeTTaIL is not merely for describing computational models, but is quite suited both to symbolic and neuromorphic AI.

8.1 Evolutionary programming: Mating Theories in Practice

We describe an approach to evolutionary programming for GSLTs. We assume an interactive GSLT, say I. That is, I enjoys a hypothesis-free (or base) reduction rule of the form $K(P, E) \rightsquigarrow K'(P', E')$. And, for simplicity, we assume this is the only rewrite of the theory. [3]

Here K is some constructor of the theory, P is a subtype of $Proc$ (the principal export of I), E is also subtype of $Proc$. We use the mnemonics P and E to remind us that P is to be thought of as "program" (aka the agent or subject of the state), and E is to be thought of as environment (the aspect of the state understood as the context in which the program or agent is operating).

Examples :

- λ-calculus:
  - $\mathsf{App}(\mathsf{Abs}([x]f), arg) \rightsquigarrow f[arg/x]$
  - $K = \mathsf{App}$, $P = \mathsf{Abs}([x]f)$, $E = arg$,
  - $P' = f[arg/x]$, $E' = \epsilon$, $K' = \Box$
- rho calculus:
  - $\mathsf{Par}(\mathsf{Recv}(x, R), \mathsf{Send}(x, Q)) \rightsquigarrow R(@Q)$
  - $K = \mathsf{Par}$, $P = \mathsf{Recv}(x, [y]R)$, $E = \mathsf{Send}(x, Q)$,
  - $P' = R[@Q/y]$, $E = 0$, $K' = \Box$
- or
  - $P = \mathsf{Send}(x, Q)$, $E = \mathsf{Recv}(x, R)$,
  - $P' = 0, E' = R(@Q)$, etc

[3] We can detect this with a spatial type on MeTTaIL theories.

Is there a cross-over operator between two interactive GSLTs, say I and J?

Suppose that $Terms(I) = Terms(J)$. For simplicity we will also suppose that $Equations(I) = Equations(J)$. However, we will revisit this. Suppose further that $base(I) = F_s(F_{1_s}(P), F_{2_s}(E)) \rightsquigarrow F_t(F_{1_t}(P'), F_{2_t}(E'))$ and $base(J) = M_s(M_{1_s}(P), M_{2_s}(E)) \rightsquigarrow M_t(M_{1_t}(P'), M_{2_t}(E'))$. Then we can define a collection of new theories, denoted $I \bowtie J$. To specify this collection we define some useful auxilliary collections.

Ed note: arities have to match.

$$
\begin{aligned}
\mathsf{src}(I \bowtie J) = \{ & \\
& F_s(F_{1_s}(P), F_{2_s}(E)), \\
& M_s(M_{1_s}(P), M_{2_s}(E)), \\
& F_s(M_{1_s}(P), F_{2_s}(E)), \\
& M_s(F_{1_s}(P), M_{2_s}(E)), \\
& F_s(F_{1_s}(P), M_{2_s}(E)), \\
& M_s(M_{1_s}(P), F_{2_s}(E)), \\
& F_s(M_{1_s}(P), M_{2_s}(E)), \\
& M_s(F_{1_s}(P), F_{2_s}(E)) \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{trgt}(I \bowtie J) = \{ & \\
& F_t(F_{1_t}(P'), F_{2_t}(E')), \\
& M_t(M_{1_t}(P'), M_{2_t}(E')), \\
& F_t(M_{1_t}(P'), F_{2_t}(E')), \\
& M_t(F_{1_t}(P'), M_{2_t}(E')), \\
& F_t(F_{1_t}(P'), M_{2_t}(E')), \\
& M_t(M_{1_t}(P'), F_{2_t}(E')), \\
& F_t(M_{1_t}(P'), M_{2_t}(E')), \\
& M_t(F_{1_t}(P'), F_{2_t}(E')) \\
& \}
\end{aligned}
$$

Finally, we can define

$$
\begin{array}{l}
I \bowtie J \\
\quad = \\
\quad \mathsf{for}(src \leftarrow \mathsf{src}(I \bowtie J);\ trgt \leftarrow \mathsf{trgt}(I \bowtie J))\ \mathsf{yield}\ \{ \\
\quad\quad \mathsf{Theory\ mkName}(src, trgt)()\{ \\
\quad\quad\quad \mathsf{Nil} \\
\quad\quad\quad \mathsf{Terms}\ \{\mathsf{Terms}(I)\} \\
\quad\quad\quad \mathsf{Equations}\ \{\mathsf{Equations}(J)\} \\
\quad\quad\quad \mathsf{Rewrites}\ \{src \rightsquigarrow trgt\} \\
\quad\quad \} \\
\quad \}
\end{array}
$$

Notice that we could have applied the same technique to contextual rewrites, as well as equations.

Now, we can describe a population, say Pop, (of theories) distributed over a namespace, say N, according to a distribution, D.

```
let islands(Pop, N, D) =
        letrec Locate(Pop, N, D)                                        =
          foldLeft(
            Pop,
            0,
            (acc, e) => {
              let (sex, chan) = (rand(0, 1), choose(N, D)) in
                let pheno =
                    match sex with {
                      case 0 ⇒ {
                        for(t@Theory {terms(e), equations(e), r} ← chan){
                          Locate(e ⋈ t, chan, Delta(chan))
                        }
                      }
                      case 1 ⇒ {chan!(e)}
                    } in
                      acc | pheno
              }
            ) in
            free(MeTTaIL) ⇒ {
              Locate(Pop, N, D)
            }
```

Note that this preserves the core action of each theory, but changes the contextual wrapper around this action. Compare this with biology. Once chemistry had kindly supplied the Krebs cycle, this core action is conserved across all life. Likewise, once biology had kindly supplied sexual reproduction, this core action is conserved across all sexually reproducing life forms.

8.2 Stochasticity and learning

In addition to the hypercube endofunctor on the category of GSLTs there is another family of endofunctors taking GSLTs to corresponding stochastic versions, both for 1- & 2-norm probability distributions, the latter effectively giving us a way to use GSLTs to write down quantum algorithms. Here we use the 1-norm probability distributions to define a generic notion of learning.

The intuition is that we can imagine a version of neural networks where the connections are between black boxes. In the ordinary case, the black box is a highly constrained computation that either fires or it doesn't based on a threshold measurement. In this case, all the intelligence is in the connectivity, rather than in the black box. We can imagine, however, making the black box more intelligent. GSLTs allow us to defined what can go in the black box, while stochasticity effectively allows us to program the connectivity. In a word, we have a compositional notion of neural networks.

To make this concrete we consider a generic learning for rholang.

$$
\begin{array}{l}
\mathsf{def}\ learn(\\
\qquad x_1, \ldots, x_n, \\
\qquad w_1, \ldots, w_n, \\
\qquad s_1, \ldots, s_m, \\
\qquad a \\
\qquad) = \{ \\
\qquad \mathsf{for}(y_1 \overset{w_1}{\leftarrow} x1 \ \& \ldots \& \ y_n \overset{w_n}{\leftarrow} x_n)\ \{ \\
\qquad\quad \mathsf{match}\ (y_1, \ldots, y_n)\ \mathsf{with} \\
\qquad\qquad \mathsf{case}\ (v1Spec1, \ldots, vnSpec1)\ \Rightarrow \{ \\
\qquad\qquad\quad a!(v_1) \\
\qquad\qquad\quad \mid learn(x_1, \ldots, x_n, s_1(w_1), \ldots, s_1(w_n), s_1, \ldots, s_m, a) \\
\qquad\qquad \} \\
\qquad\qquad \ldots \\
\qquad\qquad \mathsf{case}\ (v1Specm, \ldots, vnSpecm)\ \Rightarrow\ \{ \\
\qquad\qquad\quad a!(vm) \\
\qquad\qquad\quad \mid learn(x_1, \ldots, x_n, s_m(w_1), \ldots, s_m(w_n), s_1, \ldots, s_m, a) \\
\qquad\quad \} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}\}
\end{array}
$$

Now, under the current implementations of rholang, this will not be anything like as efficient as what's achieved by transformer based networks. However, we also have an algorithm for compiling GSLTs to hypervectors. This may yield a more efficient execution path for these kinds of compositional connection-based learning algorithms.

Again, the main point of these examples is that MeTTaIL provides a natural setting for writing AI applications that run in a decentralized setting.

9 A few use cases

9.1 Private data and models-as-a-service marketplace

As mentioned in the requirements analysis, we posit that the next generation of models will be trained on privately held data. The biggest shift, in our view, will be towards a marketplace in which companies with data and companies with model infrastructure will be able to consume one another's services. MeTTaCycle is aimed at enabling this marketplace. Beyond these we see many other use cases and list a few below.

9.2 Ocaps for DevOps PaaS

A particularly compelling application of the MeTTaCycle architecture lies in the DevOps Platform-as-a-Service (PaaS) sector, where teams seek to configure and monitor projects with minimal overhead. Developers often rely on centralized providers to share infrastructure and capabilities, which can result in vulnerability to deplatforming (for example, when payment providers or credit card companies refuse service) or require complicated permissioning frameworks.

By incorporating object-capabilities (ocaps) into a DevOps environment, MeTTaCycle could provide a robust mechanism for delegating and attenuating authorization among team members. Specifically, a lightweight plugin or extension built on top of MeTTaCycle could let organizations share their DevOps resources (such as CI/CD pipelines, artifact repositories, and monitoring dashboards) via secure tokens. This would allow fine-grained access control—even down to the level of revocable, limited-lifespan tokens for specific tasks.

This approach distinguishes itself from existing DevOps PaaS offerings in that it deploys ocaps as the fundamental security model, rather than as an add-on or ad-hoc wrapper. While no current competitor offers a sharing model based on ocaps, the development of such a plugin would likely require its own dedicated team to ensure seamless integration with existing DevOps tools.

An interesting extension of this model would be to empower autonomous AI agents with delegated ocaps for various DevOps operations. For example, an AI-driven build optimizer could be granted a strictly bounded token that allows it to adjust continuous integration parameters without risking broader system access. Such autonomous agents could also facilitate zero-downtime deployments and real-time resource scaling, all while ensuring that their authority remains precisely delimited by the ocaps architecture.

9.3 Sensors

A second use case envisions a network of industrial sensors within a smart factory, all powered by MeTTaCycle's concurrency and transactional semantics. Many factories rely on streams of data from robotic arms, assembly lines, and inventory management systems, feeding downstream analytics to optimize production. A decentralized "marketplace" for sensor data could allow different agents—both human and AI—to connect, filter, and aggregate these streams.

Using MeTTaCycle, sensors and agents become active participants in an economy of data consumption. The architecture's flexible query language can filter live streams and route relevant information to different processes for anomaly detection or predictive maintenance. The emphasis on concurrency and fine-grained access tokens could enable secure, selective data sharing among industrial partners or even across entire supply chains.

In this environment, autonomous AI agents could automatically negotiate data access or deploy new analytics modules where they detect certain operational inefficiencies. These agents might, for instance, purchase sensor data streams to train local machine-learning models or dynamically grant short-term access credentials to other agents for collaborative filtering. Crucially, the ocaps mechanism ensures they can only access precisely what they are authorized to, preventing them from overstepping their operational bounds.

9.4 Sandstorm

Sandstorm (sandstorm.org) has long provided a platform for sandboxed apps that users can self-host, ensuring data ownership. By deploying MeTTaCycle as an underlying engine for sandstorm deployments, resource owners can store, share, and collaboratively edit data in a decentralized fashion. Attenuation and delegation of capabilities would be governed by tokens: the owner (or a controlling agent) issues tokens to participants, granting privileged or time-limited access to certain documents, files, or worlds. This model aligns with sandstorm.org's mission of self-hosting and privacy protection, extending it with the added benefits of a censorship-resistant, Byzantine fault tolerant infrastructure.

Because the integration would require modifications or plugin development within the sandstorm stack itself, it might entail a dedicated development effort. The result, however, would unify the user-friendly collaboration tools sandstorm.org is known for with the secure, decentralized capabilities of MeTTaCycle.

AI agents could automate content moderation tasks, track and summarize collaborative work, or even autonomously curate shared documents for improved knowledge management. By granting these agents time-limited tokens to view and potentially modify files, users could safely outsource tasks such as grammar correction, code review, or content organization to intelligent assistants without risking full administrative access over an entire collection.

9.5 File hosting

Finally, an obvious and powerful use case for MeTTaCycle is straightforward file hosting and sharing, akin to Dropbox or Google Drive. Current commercial solutions often rely on crash fault tolerant architectures that scale well but are prone to deplatforming or security breaches at the central provider. There are also open-source alternatives, such as Tahoe-LAFS, that offer strong decentralization and encryption. MeTTaCycle naturally aligns with the design philosophy behind censorship-resistant file systems, providing concurrency, transactionality, and token-based access.

Leveraging F1R3Dr1v3's existing filesystem bridge, a user could simply drag and drop documents into a folder that is mapped onto MeTTaCycle. This approach already offers encryption at rest, with F1R3Dr1v3's environment allowing multi-party access via tokenization. The result is a file hosting platform in which censorship is far more difficult, and availability is improved through blockchain-level fault-tolerance. A key question for future work is how to best integrate advanced encryption and chunk-level redundancy strategies (similar to Tahoe-LAFS) into F1R3Dr1v3's architecture to further enhance privacy and reliability.

In this file hosting scenario, autonomous AI agents could assist with document indexing, advanced search capabilities, and automated backups. Using token-limited capabilities, agents might be granted read-only access to large sets of stored files, enabling them to provide semantic search or categorization without the risk of unauthorized edits or deletions. This delegation approach would preserve user control while unlocking AI-driven efficiencies in managing and organizing decentralized data.

10 Conclusion

If we focus just on the language design for MeTTa we see that 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
- 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.

But the goal of MeTTaCycle is to decentralize agentic computing. That means much more than simply improving the language. Specifically, it means situating a high-speed MeTTa kernel into a tokenization container, much like the EVM situates a more or less standard vm into a tokenization container; and then wrapping that in a unit of fault-tolerance, i.e. a shard governed by a consensus mechanism.

Unlike the EVM, the MeTTaCycle architecture goes a long way to demonstrating that the vast majority of the wrapper around smart contract execution need not by-hand or bespoke software. The tokenization mechanism, for example, may be derived directly from the operational semantics of a given language. Likewise, once one undertands what corresponds to a state change, even in the presence of non-determinism, the consensus mechanism may be entirely componentized. Regardless of how consensus is achieved what it must guarantee is that each node agrees each state change. But a state change is captured directly in the rewrite rules of an operational semantics. In short, any language equipped with an executable operational semantics may be turned into a smart contracting language in a completely automated fashion.

MeTTaIL provides a way to write down an executable operational semantics for all finitely presentable GSLTs. In other words, MeTTaCycle reduces these mathematical facts to practice, ushering a new wave of decentralization in which nodes may participate in multiple different networks. This promiscuity dramatically increases the opportunities for network interoperability, and allows new features to be brought from one network to another.

More generally, the ability to autogenerate the bulk of the technological apparatus that goes into decentralization lowers the bar for the development and deployment of decentralized solutions. Instead of spending years developing a smart contracting system by hand, or months forking and tweaking an existing system, the MeTTaCycle provides a collection of algorithms that can reduce this part of the effort of rolling out a new network to minutes. From an existing operational semantics spec, press a button *et voila* an efficient *tokenized* interpreter of the semantics is produced as executable code. Press another button *et voila* a spatial-behavioral type system and an efficient type-checker for it is rendered to code, thus delivering safety, liveness, and security guarantees that the current generation of smart contracts could have greatly benefitted from. Finally, merely choose the form of consensus needed for the contemplated deployment use case from a library of existing components, and *hey presto!* your network is ready for deployment.

By commoditizing these aspects of the technical challenges of providing decentralized solutions decentralization itself is further decentralized. With MeTTaCycle anyone can decentralize their offering. Reasons not to are pushed further into sociological or political spheres, not technical ones. After all, nature is both master and mistress of decentralization. Birds do it, bees do it, why can't we do it?

As we have emphasized, however, for the full potential of decentralization to be realized it must be integrated into the existing technological ecosystem so that users can use the technology as part of their day-to-day workflows, both on their local devices, and in the cloud. Currently, this does require bespoke development. For example, the existing workflows involving file systems, as well as ones involving social media are completely oblivious to tokenization. As a result, integration requires more ad hoc and bespoke solutions. Over time, however, who knows, perhaps even these integrations can be automated.

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] Stiegler, Marc (2004). *A PictureBook of Secure Cooperation.* `http://www.erights.org/talks/efun/SecurityPictureBook.pdf`.

[10] 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/`.

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

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

11 Appendix A: A Summary of MeTTaIL

This appendix was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

12 The lexical structure of MeTTaIL

12.1 Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

12.2 Literals

Character literals ⟨*Char*⟩ have the form `'`*c*`'`, where *c* is any single character.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

String literals ⟨*String*⟩ have the form `"`*x*`"`, where *x* is any sequence of any characters except `"` unless preceded by `\`.

12.3 Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in MeTTaIL are the following:

Bind	Empty	Equations
Exports	Module	NilSpace
Replacements	Rewrites	Space
Subst	Terms	Theory
as	char	coercions
comment	digit	entrypoints
eps	for	free
from	if	import
in	inf	internal
layout	let	letter
lower	nonempty	position
rules	self	separator
space	stop	sup
tail	terminator	then
theory	token	toplevel
upper		

The symbols used in MeTTaIL are the following:

{	}	(
)	=>	=
::	/\	\/
\	<|	**
|	,	:
.	_	...
@	;	!?
<-	&	?!
<=	<<-	<->
<=>	<<->>	!
!!	!$	::=
[	]	-
*	+	?
#	==	~>

12.4 Comments

Single-line comments begin with --.
Multiple-line comments are enclosed with {- and -}.

13 The syntactic structure of MeTTaIL

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Module*⟩ ::= ⟨*ListImport*⟩ `Module` ⟨*Name*⟩ `{` ⟨*ListProg*⟩ `}`

⟨*Import*⟩ ::= `import` ⟨*String*⟩ `as` ⟨*Ident*⟩
  | `import` ⟨*Ident*⟩ `from` ⟨*String*⟩

⟨*ListImport*⟩ ::= ϵ
  | ⟨*Import*⟩ ⟨*ListImport*⟩

⟨*Prog*⟩ ::= ⟨*SpaceDecl*⟩
  | ⟨*TheoryDecl*⟩
  | `space` ⟨*SpaceInst*⟩
  | `theory` ⟨*TheoryInst*⟩
  | ⟨*FactComprehension*⟩

⟨*ListProg*⟩ ::= ⟨*Prog*⟩
  | ⟨*Prog*⟩ ⟨*ListProg*⟩

⟨*SpaceInst*⟩ ::= ⟨*SpaceInst1*⟩
  | ⟨*SpaceInst*⟩ `|` ⟨*SpaceInst1*⟩

⟨*SpaceInst1*⟩ ::= ⟨*SpaceInst2*⟩
  | ⟨*SpaceInst1*⟩ `**` ⟨*SpaceInst2*⟩

⟨*SpaceInst2*⟩ ::= ⟨*SpaceInst3*⟩
  | ⟨*SpaceInst2*⟩ `<|` ⟨*SpaceInst3*⟩

⟨*SpaceInst3*⟩ ::= ⟨*SpaceInst4*⟩
  | ⟨*SpaceInst3*⟩ `\` ⟨*SpaceInst4*⟩

⟨*SpaceInst4*⟩ ::= ⟨*SpaceInst5*⟩
  | ⟨*SpaceInst4*⟩ `\/` ⟨*SpaceInst5*⟩

⟨*SpaceInst5*⟩ ::= ⟨*SpaceInst6*⟩
  | ⟨*SpaceInst5*⟩ `/\` ⟨*SpaceInst6*⟩

⟨*SpaceInst6*⟩ ::= ⟨*SpaceInst7*⟩
  | ⟨*SpaceInst6*⟩ `::` ⟨*SpaceInst7*⟩

⟨*SpaceInst7*⟩ ::= `{` ⟨*SpaceInst*⟩ `}`
  | `NilSpace`
  | ⟨*DottedPath*⟩ `(` ⟨*ListSpaceInst*⟩ `)`
  | ⟨*SpaceInst7*⟩ `=>` `{` ⟨*ListProg*⟩ `}`
  | ⟨*Ident*⟩
  | `let` ⟨*Ident*⟩ `=` ⟨*SpaceInst7*⟩ `in` `(` ⟨*SpaceInst7*⟩ `)`
  | `tail` `(` ⟨*SpaceInst7*⟩ `)`
  | `sup` `(` ⟨*SpaceInst7*⟩ `)`
  | `inf` `(` ⟨*SpaceInst7*⟩ `)`

⟨*ListSpaceInst*⟩	::=	ϵ
	|	⟨*SpaceInst*⟩
	|	⟨*SpaceInst*⟩ `,` ⟨*ListSpaceInst*⟩
⟨*TheoryInst*⟩	::=	⟨*TheoryInst1*⟩
⟨*TheoryInst1*⟩	::=	⟨*TheoryInst2*⟩
	|	⟨*TheoryInst1*⟩ `<|` ⟨*TheoryInst2*⟩
⟨*TheoryInst2*⟩	::=	⟨*TheoryInst3*⟩
	|	⟨*TheoryInst2*⟩ `\` ⟨*TheoryInst3*⟩
⟨*TheoryInst3*⟩	::=	⟨*TheoryInst4*⟩
	|	⟨*TheoryInst3*⟩ `\/` ⟨*TheoryInst4*⟩
⟨*TheoryInst4*⟩	::=	⟨*TheoryInst5*⟩
	|	⟨*TheoryInst4*⟩ `/\` ⟨*TheoryInst5*⟩
⟨*TheoryInst5*⟩	::=	⟨*TheoryInst6*⟩
	|	⟨*TheoryInst5*⟩ `Exports {` ⟨*ListExport*⟩ `}`
	|	⟨*TheoryInst5*⟩ `Replacements {` ⟨*ListReplacement*⟩ `}`
	|	⟨*TheoryInst5*⟩ `Terms {` ⟨*Grammar*⟩ `}`
	|	⟨*TheoryInst5*⟩ `Equations {` ⟨*ListEquation*⟩ `}`
	|	⟨*TheoryInst5*⟩ `Rewrites {` ⟨*ListRewriteDecl*⟩ `}`
	|	⟨*DottedPath*⟩ `(` ⟨*ListTheoryInst*⟩ `)`
	|	⟨*Ident*⟩
	|	`let` ⟨*Ident*⟩ = ⟨*TheoryInst5*⟩ `in (` ⟨*TheoryInst5*⟩ `)`
	|	`free (` ⟨*DottedPath*⟩ `)`
⟨*TheoryInst6*⟩	::=	`{` ⟨*TheoryInst*⟩ `}`
	|	`Empty`
⟨*ListTheoryInst*⟩	::=	ϵ
	|	⟨*TheoryInst*⟩
	|	⟨*TheoryInst*⟩ `,` ⟨*ListTheoryInst*⟩
⟨*SpaceDecl*⟩	::=	`Space` ⟨*Name*⟩ `(` ⟨*ListVariableDecl*⟩ `) {` ⟨*ListFact*⟩ ⟨*ListFactComprehension*⟩ `}`
⟨*TheoryDecl*⟩	::=	`Theory` ⟨*Name*⟩ `(` ⟨*ListVariableDecl*⟩ `) {` ⟨*TheoryInst*⟩ `}`
⟨*VariableDecl*⟩	::=	⟨*Ident*⟩ `:` ⟨*DottedPath*⟩
⟨*ListVariableDecl*⟩	::=	ϵ
	|	⟨*VariableDecl*⟩
	|	⟨*VariableDecl*⟩ `,` ⟨*ListVariableDecl*⟩
⟨*DottedPath*⟩	::=	⟨*Ident*⟩
	|	⟨*Ident*⟩ `.` ⟨*DottedPath*⟩

⟨*Name*⟩ ::= `_`
| ⟨*Ident*⟩

⟨*ListName*⟩ ::= ϵ
| ⟨*Name*⟩
| ⟨*Name*⟩ `,` ⟨*ListName*⟩

⟨*NameRemainder*⟩ ::= `... @` ⟨*Ident*⟩
| ϵ

⟨*Fact*⟩ ::= ⟨*Item*⟩

⟨*ListFact*⟩ ::= ⟨*Fact*⟩
| ⟨*Fact*⟩ `;` ⟨*ListFact*⟩

⟨*FactComprehension*⟩ ::= `for (` ⟨*ListReceipt*⟩ `)` ⟨*SpaceInst7*⟩
| ⟨*Name*⟩ ⟨*Send*⟩ `(` ⟨*ListFact*⟩ `)`
| ⟨*Name*⟩ ⟨*Send*⟩ `( )`
| ⟨*Name*⟩ `!? (` ⟨*ListFact*⟩ `)` ⟨*SynchSendCont*⟩
| ⟨*Name*⟩ `!? ( )` ⟨*SynchSendCont*⟩

⟨*ListFactComprehension*⟩ ::= ϵ
| ⟨*FactComprehension*⟩
| ⟨*FactComprehension*⟩ `,` ⟨*ListFactComprehension*⟩

⟨*Receipt*⟩ ::= ⟨*ReceiptLinearImpl*⟩
| ⟨*ReceiptRepeatedImpl*⟩
| ⟨*ReceiptPeekImpl*⟩
| ⟨*ReceiptSymmLinearImpl*⟩
| ⟨*ReceiptSymmRepeatedImpl*⟩
| ⟨*ReceiptSymmPeekImpl*⟩

⟨*ListReceipt*⟩ ::= ⟨*Receipt*⟩
| ⟨*Receipt*⟩ `;` ⟨*ListReceipt*⟩

⟨*ReceiptLinearImpl*⟩ ::= ⟨*ListLinearBind*⟩

⟨*LinearBind*⟩ ::= ⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<-` ⟨*NameSource*⟩

⟨*ListLinearBind*⟩ ::= ⟨*LinearBind*⟩
| ⟨*LinearBind*⟩ `&` ⟨*ListLinearBind*⟩

⟨*NameSource*⟩ ::= ⟨*Name*⟩
| `self`
| ⟨*Name*⟩ `?!`
| ⟨*Name*⟩ `!? (` ⟨*ListFact*⟩ `)`

⟨*ReceiptRepeatedImpl*⟩ ::= ⟨*ListRepeatedBind*⟩

⟨*RepeatedBind*⟩	::=	⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<=` ⟨*Name*⟩
⟨*ListRepeatedBind*⟩	::=	⟨*RepeatedBind*⟩
	|	⟨*RepeatedBind*⟩ `&` ⟨*ListRepeatedBind*⟩
⟨*ReceiptPeekImpl*⟩	::=	⟨*ListPeekBind*⟩
⟨*PeekBind*⟩	::=	⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<<-` ⟨*Name*⟩
⟨*ListPeekBind*⟩	::=	⟨*PeekBind*⟩
	|	⟨*PeekBind*⟩ `&` ⟨*ListPeekBind*⟩
⟨*ReceiptSymmLinearImpl*⟩	::=	⟨*ListLinearBindSymm*⟩
⟨*LinearBindSymm*⟩	::=	⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<->` ⟨*NameSource*⟩
⟨*ListLinearBindSymm*⟩	::=	⟨*LinearBindSymm*⟩
	|	⟨*LinearBindSymm*⟩ `&` ⟨*ListLinearBindSymm*⟩
⟨*ReceiptSymmRepeatedImpl*⟩	::=	⟨*ListRepeatedBindSymm*⟩
⟨*RepeatedBindSymm*⟩	::=	⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<=>` ⟨*Name*⟩
⟨*ListRepeatedBindSymm*⟩	::=	⟨*RepeatedBindSymm*⟩
	|	⟨*RepeatedBindSymm*⟩ `&` ⟨*ListRepeatedBindSymm*⟩
⟨*ReceiptSymmPeekImpl*⟩	::=	⟨*ListPeekBindSymm*⟩
⟨*PeekBindSymm*⟩	::=	⟨*ListName*⟩ ⟨*NameRemainder*⟩ `<<->>` ⟨*Name*⟩
⟨*ListPeekBindSymm*⟩	::=	⟨*PeekBindSymm*⟩
	|	⟨*PeekBindSymm*⟩ `&` ⟨*ListPeekBindSymm*⟩
⟨*SynchSendCont*⟩	::=	`.`
	|	`;` ⟨*SpaceInst*⟩
⟨*Send*⟩	::=	`!`
	|	`!!`
	|	`!$`
⟨*Export*⟩	::=	⟨*Cat*⟩
⟨*ListExport*⟩	::=	ϵ
	|	⟨*Export*⟩ `;` ⟨*ListExport*⟩
⟨*Replacement*⟩	::=	⟨*IntList*⟩ ⟨*Label*⟩ `.` ⟨*Cat*⟩ `=>` ⟨*Def*⟩

⟨*ListReplacement*⟩ ::= ϵ
  | ⟨*Replacement*⟩ `;` ⟨*ListReplacement*⟩

⟨*Grammar*⟩ ::= ⟨*ListDef*⟩

⟨*ListDef*⟩ ::= ϵ
  | ⟨*Def*⟩ `;` ⟨*ListDef*⟩

⟨*ListItem*⟩ ::= ϵ
  | ⟨*Item*⟩ ⟨*ListItem*⟩

⟨*Def*⟩ ::= ⟨*Label*⟩ `.` ⟨*Cat*⟩ `::=` ⟨*ListItem*⟩
  | `comment` ⟨*String*⟩
  | `comment` ⟨*String*⟩ ⟨*String*⟩
  | `internal` ⟨*Label*⟩ `.` ⟨*Cat*⟩ `::=` ⟨*ListItem*⟩
  | `token` ⟨*Ident*⟩ ⟨*Reg*⟩
  | `position token` ⟨*Ident*⟩ ⟨*Reg*⟩
  | `entrypoints` ⟨*ListIdent*⟩
  | `separator` ⟨*MinimumSize*⟩ ⟨*Cat*⟩ ⟨*String*⟩
  | `terminator` ⟨*MinimumSize*⟩ ⟨*Cat*⟩ ⟨*String*⟩
  | `coercions` ⟨*Ident*⟩ ⟨*Integer*⟩
  | `rules` ⟨*Ident*⟩ `::=` ⟨*ListRHS*⟩
  | `layout` ⟨*ListString*⟩
  | `layout stop` ⟨*ListString*⟩
  | `layout toplevel`

⟨*Item*⟩ ::= ⟨*String*⟩
  | ⟨*Cat*⟩
  | `( Bind` ⟨*Ident*⟩ ⟨*IntList*⟩ `)`

⟨*Cat*⟩ ::= `[` ⟨*Cat*⟩ `]`
  | ⟨*Ident*⟩
  | ⟨*Ident*⟩ `.` ⟨*Cat*⟩

⟨*Label*⟩ ::= ⟨*LabelId*⟩
  | ⟨*LabelId*⟩ ⟨*ListProfItem*⟩
  | ⟨*LabelId*⟩ ⟨*LabelId*⟩ ⟨*ListProfItem*⟩
  | ⟨*LabelId*⟩ ⟨*LabelId*⟩

⟨*LabelId*⟩ ::= ⟨*Ident*⟩
  | `_`
  | `[ ]`
  | `( : )`
  | `( : [ ] )`

⟨*ProfItem*⟩ ::= `( [` ⟨*ListIntList*⟩ `] , [` ⟨*ListInteger*⟩ `] )`

⟨*IntList*⟩ ::= `[` ⟨*ListInteger*⟩ `]`

$\langle \mathit{ListInteger} \rangle$ ::= ϵ
| $\langle \mathit{Integer} \rangle$
| $\langle \mathit{Integer} \rangle$ `,` $\langle \mathit{ListInteger} \rangle$

$\langle \mathit{ListIntList} \rangle$ ::= ϵ
| $\langle \mathit{IntList} \rangle$
| $\langle \mathit{IntList} \rangle$ `,` $\langle \mathit{ListIntList} \rangle$

$\langle \mathit{ListProfItem} \rangle$ ::= $\langle \mathit{ProfItem} \rangle$
| $\langle \mathit{ProfItem} \rangle$ $\langle \mathit{ListProfItem} \rangle$

$\langle \mathit{ListString} \rangle$ ::= $\langle \mathit{String} \rangle$
| $\langle \mathit{String} \rangle$ `,` $\langle \mathit{ListString} \rangle$

$\langle \mathit{ListRHS} \rangle$ ::= $\langle \mathit{RHS} \rangle$
| $\langle \mathit{RHS} \rangle$ | $\langle \mathit{ListRHS} \rangle$

$\langle \mathit{RHS} \rangle$ ::= $\langle \mathit{ListItem} \rangle$

$\langle \mathit{MinimumSize} \rangle$ ::= `nonempty`
| ϵ

$\langle \mathit{Reg2} \rangle$ ::= $\langle \mathit{Reg2} \rangle$ $\langle \mathit{Reg3} \rangle$
| $\langle \mathit{Reg3} \rangle$

$\langle \mathit{Reg1} \rangle$ ::= $\langle \mathit{Reg1} \rangle$ | $\langle \mathit{Reg2} \rangle$
| $\langle \mathit{Reg2} \rangle$ $-$ $\langle \mathit{Reg2} \rangle$
| $\langle \mathit{Reg2} \rangle$

$\langle \mathit{Reg3} \rangle$ ::= $\langle \mathit{Reg3} \rangle$ `*`
| $\langle \mathit{Reg3} \rangle$ $+$
| $\langle \mathit{Reg3} \rangle$ `?`
| `eps`
| $\langle \mathit{Char} \rangle$
| `[` $\langle \mathit{String} \rangle$ `]`
| `{` $\langle \mathit{String} \rangle$ `}`
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| `(` $\langle \mathit{Reg} \rangle$ `)`

$\langle \mathit{Reg} \rangle$ ::= $\langle \mathit{Reg1} \rangle$

$\langle \mathit{ListIdent} \rangle$ ::= $\langle \mathit{Ident} \rangle$
| $\langle \mathit{Ident} \rangle$ `,` $\langle \mathit{ListIdent} \rangle$

$\langle \mathit{Equation} \rangle$ ::= `if` $\langle \mathit{Ident} \rangle$ `#` $\langle \mathit{Ident} \rangle$ `then` $\langle \mathit{Equation} \rangle$
| $\langle \mathit{AST} \rangle$ $==$ $\langle \mathit{AST} \rangle$

$$
\begin{array}{lll}
\langle ListEquation \rangle & ::= & \epsilon \\
 & | & \langle Equation \rangle \ \texttt{;} \ \langle ListEquation \rangle
\end{array}
$$

$$
\langle RewriteDecl \rangle \ ::= \ \langle Ident \rangle \ \texttt{:} \ \langle Rewrite \rangle
$$

$$
\begin{array}{lll}
\langle ListRewriteDecl \rangle & ::= & \epsilon \\
 & | & \langle RewriteDecl \rangle \\
 & | & \langle RewriteDecl \rangle \ \texttt{;} \ \langle ListRewriteDecl \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle Rewrite \rangle & ::= & \langle AST \rangle \ \texttt{\~{}>} \ \langle AST \rangle \\
 & | & \texttt{let} \ \langle Hypothesis \rangle \ \texttt{in} \ \langle Rewrite \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle AST \rangle & ::= & \texttt{(} \ \texttt{Subst} \ \langle AST \rangle \ \langle AST \rangle \ \langle Ident \rangle \ \texttt{)} \\
 & | & \langle Ident \rangle \\
 & | & \texttt{(} \ \langle Ident \rangle \ \langle ListAST \rangle \ \texttt{)}
\end{array}
$$

$$
\begin{array}{lll}
\langle ListAST \rangle & ::= & \epsilon \\
 & | & \langle AST \rangle \ \langle ListAST \rangle
\end{array}
$$

$$
\langle Hypothesis \rangle \ ::= \ \langle Ident \rangle \ \texttt{\~{}>} \ \langle Ident \rangle
$$

14 Appendix B: Examples of MeTTaIL

```
//UnivAlg.module file contents
Module UnivAlg {
  Theory TwoPoint() {
    Empty
      Exports {
        Proc;
      }
      Terms {
        One . Proc ::= "1" ;
        Two . Proc ::= "2" ;
      }
  }

  Theory Monoid() {
    Empty
      Exports {
        Proc;
      }
      Terms {
        One . Proc ::= "1" ;
        Mult . Proc ::= "(" Proc "*" Proc ")" ;
      }
      Equations {
```

```
      (Mult (Mult x y) z) == (Mult x (Mult y z));
      (Mult x (One)) == x;
      (Mult (One) x) == x;
    }
}

Theory CommutativeMonoid(m: u.Monoid) {
  m
    Replacements {
      [] One . Proc => Zero . Proc ::= "0" ;
      [0, 1] Mult . Proc => Plus . Proc ::= "(" Proc "+" Proc ")" ;
    }
    Equations {
      (Plus x y) == (Plus y x);
    }
}

Theory Rig(add: CommutativeMonoid, mult: Monoid) {
  {add \/ mult}
    Equations {
      (Mult x (Plus y z)) == (Plus (Mult x y) (Mult x z));
      (Mult (Plus x y) z) == (Plus (Mult x z) (Mult y z));
      (Mult x (Zero)) == (Zero);
      (Mult (Zero) x) == (Zero);
    }
}

Theory Group(m: Monoid) {
  m
    Terms {
      Inv . Proc ::= "inv" "(" Proc ")" ;
    }
    Equations {
      (Mult x (Inv x)) == (One);
      (Inv (Inv x)) == x;
      (Inv (Mult x y)) == (Mult (Inv y) (Inv x));
    }
}

Theory AbelianGroup(g: Group, c: CommutativeMonoid) {
  g
    Replacements {
      [] One . Proc => Zero . Proc ::= "0" ;
      [0, 1] Mult . Proc => Plus . Proc ::= "(" Proc "+" Proc ")" ;
      [0] Inv . Proc => Neg . Proc ::= "(" "-" Proc ")" ;
    } \/ c
```

```
  }

  Theory Ring(r: Rig, add: AbelianGroup) {
    r \/ add
  }

  theory TwoPoint() \/ Monoid()
}

//Rholang.module file contents

import Monoid from "UnivAlg.module"

Module Rholang {
  Theory ParMonoid(cm: u.CommutativeMonoid) {
    cm
      Replacements {
        [] Zero.Proc => PZero.Proc ::= "0";
        [0, 1] Plus.Proc => PPar.Proc ::= "(" Proc "|" Proc ")";
      }
      Rewrites {
        RPar1 : let Src ~> Tgt in
                ( PPar Src Q ) ~> ( PPar Tgt Q ) ;
        RPar2 : let Src1 ~> Tgt1 in
                let Src2 ~> Tgt2 in
                ( PPar Src1 Src2 ) ~> ( PPar Tgt1 Tgt2 ) ;
      }
  }

  Theory NewReplCalc(pm: ParMonoid) {
    pm
      Exports {
        Name ;
      }
      Terms {
        PRepl . Proc ::= "!" Proc ;
        PNew  . Proc ::= "new" (Bind Name [1]) "in" Proc ;
      }
      Equations {
        if x # Q then
          ( PPar ( PNew x P ) Q ) == ( PNew x ( PPar P Q ) ) ;
        ( PNew x ( PNew x P ) ) == ( PNew x P ) ;
        ( PNew x ( PNew y P ) ) == ( PNew y ( PNew x P ) ) ;
        ( PRepl P ) == ( PPar P ( PRepl P ) ) ;
      }
      Rewrites {
```

```
      RNew  : let Src ~> Tgt in
              (PNew x Src) ~> (PNew x Tgt) ;
    }
}

Theory QuoteDropCalc(pm: ParMonoid) {
  pm
    Exports {
      Name ;
    }
    Terms {
      PDrop . Proc ::= "*" Name ;
      NQuote . Name ::= "@" Proc ;
    }
    Equations {
      ( NQuote ( PDrop N ) ) == N ;
      ( PDrop ( NQuote P ) ) == P ;
    }
}

Theory RhoCalc(qd: QuoteDropCalc) {
  qd
    Terms {
      PSend . Proc ::= Name "!" "(" Proc ")" ;
      PRecv . Proc ::= "for" "(" (Bind Name [2]) "<-" Name ")" "{" Proc "}" ;
    }
    Rewrites {
      RComm : ( PPar ( PRecv y x P) ( PSend x Q ) ) ~> ( Subst P ( NQuote Q ) y ) ;
    }
}

Theory Rholang(nr: NewReplCalc, r: RhoCalc) {
  nr \/ r
}

Theory FreeRholang() {
  let m = Monoid() in (
  let cm = CommutativeMonoid(m) in (
  let pm = ParMonoid(cm) in (
  let qd = QuoteDropCalc(pm) in (
  let nr = NewReplCalc(pm) in (
  let rc = Rhocalc(qd) in (
  let rl = Rholang(nr, rc) in (
  r1
  )))))))
}
```

```
  -- theory free(Monoid)
  -- theory free(Rholang)
  -- theory let x = Empty Exports { Foo; } in (x Exports { Bar; })
  theory Monoid()
}
```

15 Appendix C: Inference rules expected to be generated by the conjectured hypercube functor acting on the GSLT for RHO calculus

1. Axioms, Structural Types

$$\forall \text{ shapes } T \text{ in GSLT}, *^T, \Box^T : P.$$

$$\frac{}{\vdash *^T : \Box^T}$$

2. Start

$$\frac{\Gamma \vdash A : s^T}{\Gamma,\, x : A \vdash x : A}$$

3. Weakening

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s^T}{\Gamma,\, x : C \vdash A : B}$$

4. Lambda Theory

(a) Dependent Product

$$\Lambda : P \times (P \to P) \to P.$$

$$\frac{\Gamma \vdash A : s_1^T \qquad \Gamma,\, x : A \vdash B : s_2^T}{\Gamma \vdash \Lambda(A,\, \lambda x.B) : s_2^{T \to T'}}$$

(b) Abstraction

$$\frac{\Gamma \vdash A : s_1^T \qquad \Gamma,\, x : A \vdash B : s_2^T \qquad \Gamma,\, x : A \vdash C : B}{\Gamma \vdash \lambda x.C : \Lambda(A,\, \lambda x.B)}$$

(c) Application 1

$$\frac{\Gamma \vdash A : s_1^T \qquad \Gamma,\, x : A \vdash B : s_2^T \qquad \Gamma \vdash D : A}{\Gamma \vdash (\lambda x.B\, D) : s_2^T}$$

(d) Application 2

$$\frac{\Gamma \vdash C : \Lambda(A,\, B) \qquad \Gamma \vdash D : A}{\Gamma \vdash (C\, D) : (B\, D)}$$

5. Structural Types and Extending Term Constructors with Exponential Parameters

$$\frac{}{\vdash 0 : s^P}$$

$$\frac{}{\vdash 0 : 0}$$

Here the two sorts are forced to be equal because $|$ is commutative.

$$\frac{\Gamma \vdash A : s^P \qquad \Gamma \vdash B : s^P}{\Gamma \vdash A \mid B : s^P}$$

$$\frac{\Gamma \vdash A : s^P \qquad \Gamma \vdash B : A \qquad \Gamma \vdash C : s^P \qquad \Gamma \vdash D : C}{\Gamma \vdash B \mid D : A \mid C}$$

The name x has to be part of the type because the left-hand side of the comm rule duplicates x. It's not yet clear what forces $x!(B)$ to be of the same sort as x. It might be for the same reason, or it might make sense with either choice; if the latter, then there would be a different hypercube functor per choice.

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P}{\Gamma \vdash x!(B) : s_1^P}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P \qquad \Gamma \vdash Q : B}{\Gamma \vdash x!(Q) : x!(B)}$$

The functor adds an extra parameter to for to track the type of the bound variable.

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^N \qquad \Gamma,\, y : B \vdash C : s_3^P}{\Gamma \vdash \mathrm{for(x, B, \lambda y.C) : s_1^P}}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^N \qquad \Gamma, y : B \vdash C : s_3^P \qquad \Gamma, y : B \vdash D : C}{\Gamma \vdash \mathrm{for(x, B, \lambda y.D) : for(x, B, \lambda y.C)}}$$

$$\frac{\Gamma \vdash A : s^N}{\Gamma \vdash *A : s^P}$$

$$\frac{\Gamma \vdash A : s^N \qquad \Gamma \vdash x : A}{\Gamma \vdash *x : *A}$$

$$\frac{\Gamma \vdash A : s^P}{\Gamma \vdash @A : s^N}$$

$$\frac{\Gamma \vdash A : s^P \qquad \Gamma \vdash Q : A}{\Gamma \vdash @Q : @A}$$

6. App-like Rules for Rewrites with Substitution in RHS

The base rewrite rule is

$$\mathrm{comm:} \quad \mathrm{x!(C) \mid for(x, A, B) \rightsquigarrow \big(B\ @C\big).}$$

The for term constructor takes a lambda term B as a parameter that's evaluated on the RHS, so we get the following rules:

(a) Empty Context

$$\Diamond : P \to P.$$

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash A : \Diamond B}$$

$$\frac{\Gamma \vdash C : \mathrm{for(x, A, B)} \qquad \Gamma \vdash D : *A}{\Gamma \vdash C \mid x!(D) : \Diamond(B\ @D)}$$

(b) LHS Context

$$\frac{\Gamma \vdash C : \mathrm{for(x, A, B)} \qquad \Gamma \vdash D : *A}{\Gamma \vdash C : \mathrm{\langle [for(x, A, B)] \mid x!(D) \rangle (B\ @D)}}$$

7. Modalities for Non-Exponential Positions in Base Rules

$$\frac{\Gamma \vdash A : s_1^P \qquad \Gamma \vdash C : s_2^{P \to P} \qquad \Gamma \vdash K : C}{\Gamma \vdash \langle K \rangle A : s_1^P}$$

$$\frac{\Gamma \vdash A : s_1^P \qquad \Gamma \vdash C : s_2^{P \to P} \qquad \Gamma \vdash K : C \qquad \Gamma \vdash B : \langle K \rangle A}{\Gamma \vdash (K\,B) : \Diamond A}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P \qquad \Gamma \vdash Q : B \qquad \Gamma,\, y : @B \vdash C : s_3^P \qquad \Gamma,\, y : @B \vdash D : C}{\Gamma \vdash x!(Q) \mid \mathrm{for(x, @B, \lambda y.D)} : \Diamond\big(\mathrm{(\lambda y.C)\;@Q}\big)}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P \qquad \Gamma \vdash Q : B \qquad \Gamma,\, y : @B \vdash C : s_3^P \qquad \Gamma,\, y : @B \vdash D : C}{\Gamma \vdash x!(Q) : \langle [\, x!(B)\,] \mid \mathrm{for(x, @B, \lambda y.D)} \rangle \big(\mathrm{(\lambda y.C)\;@Q}\big)}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P \qquad \Gamma \vdash Q : B \qquad \Gamma,\, y : @B \vdash C : s_3^P \qquad \Gamma,\, y : @B \vdash D : C}{\Gamma \vdash \mathrm{for(x, @B, \lambda y.D)} : \langle \mathrm{x!(Q)} \mid [\, \mathrm{for(x, @B, \lambda y.C)}\,] \rangle \big(\mathrm{(\lambda y.C)\;@Q}\big)}$$

$$\frac{\Gamma \vdash A : s_1^N \qquad \Gamma \vdash x : A \qquad \Gamma \vdash B : s_2^P \qquad \Gamma \vdash Q : B \qquad \Gamma,\, y : @B \vdash C : s_3^P \qquad \Gamma,\, y : @B \vdash D : C}{\Gamma \vdash \lambda y.D : \langle x!(Q) \mid \mathrm{for(x, @B, [\, \Lambda(@B, \lambda y.C)\,])} \rangle \big(\mathrm{(\lambda y.C)\;@Q}\big)}$$

8. Conv-like Rules

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : A \qquad \Gamma \vdash \rho : B \rightsquigarrow B' \qquad \Gamma \vdash B' : A'}{\Gamma \vdash B : \Diamond A'}$$

$$\frac{\Gamma \vdash A : s_1^P \qquad \Gamma \vdash B : \Diamond A \qquad \Gamma \vdash C : \langle K[A] \rangle D}{\Gamma \vdash (K\,B) : \Diamond\Diamond D}$$

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash (K\,A) : (K\,B)}$$

$$\Diamond^* : P \to P.$$

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash A : \Diamond^* B}$$

$$\frac{\Gamma \vdash A : \Diamond\Diamond^* B}{\Gamma \vdash A : \Diamond^* B}$$

$$\frac{\Gamma \vdash A : \Diamond^*\Diamond B}{\Gamma \vdash A : \Diamond^* B}$$

$$\frac{\Gamma \vdash A : \Diamond^* \Diamond^* B}{\Gamma \vdash A : \Diamond^* B}$$

$$\frac{\Gamma \vdash A : s_1^P \qquad \Gamma \vdash B : \Diamond^* A \qquad \Gamma \vdash C : \langle K[A] \rangle D}{\Gamma \vdash (K\, B) : \Diamond^* D}$$