MeTTaMath: Integrating Formal Verification into an AGI Cognitive Architecture via the MeTTa language

From Metamath proof checking to AGI-native verified reasoning

Zarathustra Amadeus Goertzel

September 1, 2025

Czech Institute of Informatics, Robotics and Cybernetics

Motivating Question: How should verified reasoning be integrated into AGIs?

- The ITP model: lean proof kernel that can outsource proof-search to ATPs.
  - Is it important to have proofs in the cognitive language of the AGI system?
  - This may reduce translation errors between systems, etc.
- I'm working with HYPERON—an AGI framework fostering *cognitive synergy* among diverse components using shared knowledge representations.
- METTA: gradually-typed meta-programming language; programming as **pattern matching & rewriting over metagraphs**.
- If wishing to do inference control experiments as Nil suggests with PLN, which mathematical library should be used?
- (I am neither an expert in MeTTa nor Metamath: there exist educational motives.)

Why Metamath?

- Ultra-minimal proof language with a **single core rule: substitution**.
- Existing tiny Python verifier (`mmverify.py`) — I'd like to minimize work!
- Good alignment with MeTTa: the proof structure may be *natively* similar in MeTTa.

MeTTaMath: State of the Project

- Metamath verifier implemented in MeTTa, as a deep embedding[1].
- Small Metamath tests passed.
- Simple `demo0.mm` (proving $t = t$) passed through the backward chainer: `demo0_bc.metta`.

[1] Deep = object language as data. Shallow = map constructs directly to host semantics.

Implementation Sketch: Verifier Overview

- `mmverify.py` parses the METAMATH file sequentially and maintains a **frames stack** (scope) with:
  ▷ Active variable symbols, active floating hypotheses (≈ type decls), essential hypotheses (assumptions), and disjoint-variable constraints (≈ to manage variable scoping).
- Constants, assertions, and proven statements are indexed by label.
- Verification uses a stack to construct the assertion.
- Verification step (hypotheses): push onto the stack.
- Verification step (assertions):
  1. Treat each proof step to construct the target assertion via a substitution stack:
     (a) Construct the substitution from `f_hyps`;
     (b) Check that the substituted `e_hyps` match the assertion's `e_hyps`;
     (c) Check disjoint-variable constraints: if $d(x, y)$, then
     - $V(\sigma(x)) \cap V(\sigma(y)) = \varnothing$;
     - $\forall x_i \in V(\sigma(x)),\ y_i \in V(\sigma(y)),\ d(x_i, y_i)$.
  2. Push the σ-substituted conclusion onto the stack.

Implementation Sketch: Verifier Overview (parsing)

- MeTTa HE 0.2.6 is very slow and the string operations currently go through Python, so I didn't bother implementing the parsing.
- The MM.read() function generally looks as follows:

```
        mettarl(f'!(add_f {mettify(label)} {mettify(stmt[0])} {mettify(stmt[1])} {len(self.fs)})')
        self.add_f(stmt[0], stmt[1], label)
        self.labels[label] = ('$f', [stmt[0], stmt[1]])
        label = None
    elif tok == '$e':
        if not label:
            raise MMError('$e must have label')
        stmt = self.read_non_p_stmt(tok, toks)
        mettarl(f'!(add_e {mettify(label)} {mettify(stmt)} {len(self.fs)})')
        self.fs.add_e(stmt, label)
        self.labels[label] = ('$e', stmt)
        label = None
    elif tok == '$a':
        if not label:
            raise MMError('$a must have label')
        stmt = self.read_non_p_stmt(tok, toks) # Just less-compact
        mettarl(f'!(add_a {mettify(label)} {mettify(stmt)})')
        dvs, f_hyps, e_hyps, stmt = self.fs.make_assertion(stmt) # make_assertion(self.read_non_p_stmt(tok, toks))
        self.labels[label] = ('$a', (dvs, f_hyps, e_hyps, stmt))
```

MeTTa Basics Interlude

- Everything is an **Atom** (of metatypes: Symbol, Variable, Ground and Expression)
- To me it feels like a mix of declarative and functional programming.
- Data live in **spaces**; which can be queried with **match** and **unify**, and one can **chain** on the results.
- Rewriting: `(= (lhs) rhs)` defines reduction rules rules; matching binds variables.
- Results are *superpositions* of matches; non-determinism is the default.

MeTTa Basics Interlude

```
> ;; Bind the token $subst to a new spcae
!(bind! &subst (new-space))

;; Insert substitution pairs as atoms into the space
!(add-atom &subst ("P" ("(" "t" "+" "0" ")" "=" "t")))
!(add-atom &subst ("Q" ("t" "=" "t")))

;; Now the space can be matched
!(match &subst ("P" $rhs) $rhs)

[()]
[()]
[()]
[("(" "t" "+" "0" ")" "=" "t")]
```

MeTTa Basics Interlude

```
> ;; Define tokenwise substitution over a 'string' using unify
(= (apply_subst_tok $space $tok)
   (unify $space ($tok $rhs) $rhs $tok))

(= (apply_subst $stmt $space)
   (map-atom $stmt $tok (apply_subst_tok $space $tok)))

;; Run it on a Metamath-like token list
!(apply_subst ("|-" "(" "P" "->" "Q" ")") &subst)
[("|-" "(" ("(" "t" "+" "0" ")" "=" "t") "->" ("t" "=" "t") ")")]
```

MeTTa Data Structures Used

- I use a &stack *space* for the stack.
- I use a &sp *state* for the stack pointer.
- I use $subst spaces to build up substitution dictionaries.
- I use the &kb space for everything else:
  - The *labels* of $f, $e, $a, and $p statements.
  - The *frame stack* by adding *(FSDepth $d)* atoms to expressions on the stack[2].

[2] I confess to doing embarrassingly little effort to optimize for performance rather than correctness, unless, however, it was too excruciatingly slow to even do small examples.

Implementation Sketch: Verifier Overview (essential hypotheses)

- **add_e** adds an essential hypothesis statement to the frame and the list of e_hyps at that frame.

```
(= (add_e $label $stmt $level)
  (let*
    (
      ($elist_entry (to-list ($stmt)))
      (() (unify &kb (EList (FSDepth $level) $elist)
        (let $elist' (append $elist $elist_entry)
          (update-atom &kb (EList (FSDepth $level) $elist) (EList (FSDepth $level) $elist')))
        (add-atom &kb (EList (FSDepth $level) $elist_entry))))
        (() (println! ("add essential hypothesis:" (label $label) (Statement $stmt) (level $level))))
    )
      (add-atom &kb ( (Label $label) EHyp (FSDepth $level) ( (Statement $stmt) (Type "$e") )))
    )
)
```

Implementation Sketch: Verifier Overview (disjoint variables)

- **add_d** takes a variable list and adds each new oriented pair to the frame (via &kb).

```
(= (add_dv_pair_if_fresh $x $y $level)
  (if (== $x $y) ()
    (let ($ox $oy) (orient_pair $x $y)
      (unify &kb (DVar ($ox $oy) (FSDepth $level) (Type "$d"))
        ()
        (add-atom &kb (DVar ($ox $oy) (FSDepth $level) (Type "$d") ))))))

(= (add_d $varlist $level)
  (map-atom $varlist $x
    (map-atom $varlist $y
      (add_dv_pair_if_fresh $x $y $level))))
```

Implementation Sketch: Verifier Overview (floating hypotheses)

- **add_f** registers a floating hypothesis at the current frame depth and adds it to the list of f_hyps at that frame.
- Checks that the *var* and *typecode* are declared, and that the var isn't assigned to any other typecode.

```
(= (add_f $label $typecode $var $level)
  (unify &kb (Var $var $_ (Type "$v"))
    (unify &kb (Constant $typecode (Type "$c"))
      (unify &kb ( (Label $label') FHyp (FSDepth $FSDepth) ( (Typecode $typecode') (FVar $var) (Type "$f") ) )
        (Error ( (Label $label') (Typecode $typecode') (Var $var) ) "Var in $f already typed by an active $f-statement." )
        (let*
          (
            ($flist_entry (to-list (($typecode $var))))
            (() (unify &kb (FList (FSDepth $level) $flist)
              (let $flist' (append $flist $flist_entry) (update-atom &kb (FList (FSDepth $level) $flist) (FList (FSDepth $level) $flist')))
              (add-atom &kb (FList (FSDepth $level) $flist_entry))))
          )
          (add-atom &kb ( (Label $label) FHyp (FSDepth $level) ( (Typecode $typecode) (FVar $var) (Type "$f") )))
        )
      )
      (Error (Constant $typecode) "Typecode in $f not declared."))
    (Error (Var $var) "Var in $f not declared."))
)
```

Implementation Sketch: Verifier Overview (axiomatic assertions)

- **add_a** makes an assertion based on the current frame scopes and the statement.

```
(= (add_a $label $stmt)
  (let*
    (
      (() (println! ("make_assertion" $label - $stmt)))
      (( (DVars $dvs) (FHyps $f_hyps) (EHyps $e_hyps) (Statement $stmt) ) (make_assertion $stmt))
      (() (println! ("gathered assertion data:" $dvs $f_hyps $e_hyps $stmt)))
    )
    (add-atom &kb ( (Label $label) Assertion ( (DVars $dvs) (FHyps $f_hyps) (EHyps $e_hyps) (Statement $stmt) (Type "$a") )))))
```

Implementation Sketch: Verifier Overview (provable assertions)

- **add_p** does the same as add_a after verifying the proof.

```
(= (add_p $label $stmt $proof $verify_proofs)
  (let*
    (
      (() (println! ""))
      (() (println! (Verifying conclusion ($label) $stmt with proof $proof)))
      (() (if $verify_proofs (verify $proof $stmt) ()))
      (() (println! ("make_assertion" $label - $stmt))) ;; This could just call add_a.
      (( (DVars $dvs) (FHyps $f_hyps) (EHyps $e_hyps) (Statement $stmt) ) (make_assertion $stmt))
      (() (println! ("gathered assertion data:" $dvs $f_hyps $e_hyps $stmt)))
    )
      (add-atom &kb ( (Label $label) Proof ( (DVars $dvs) (FHyps $f_hyps) (EHyps $e_hyps) (Statement $stmt) (Type "$p") (ProofSequence $proof) )))))
```

Implementation Sketch: Verifier Overview (make assertion)

Collect in scope e_hyps, mark mandatory vars, and their DVs and f_hyps.

```
(= (make_assertion $stmt) (let* (
      ($e_hyps_lists (matchc &kb (EList (FSDepth $level) $elist) ($level $elist)))
      ($e_levels (collapse (match-atom' $e_hyps_lists ($l $_) $l)))
      ($e_max_level (if (== $e_levels (())) 0 (max-atom $e_levels)))
      ($e_hyps_list (collect_lists_by_depth $e_hyps_lists 1 $e_max_level Nil))
      ($e_hyps_toks (from-list (flatten-list $e_hyps_list)))
      ($_0 (map-atom $e_hyps_toks $tok (add_mand_var $tok)))
      ($_1 (map-atom $stmt $tok (add_mand_var $tok)))
      ($mand_vars (matchc &kb (MandVar $var) $var))
      ($dvs (matchc &kb (DVar ($x $y) $_ (Type "$d")) (unify &kb (MandVar $x) (unify &kb (
     MandVar $y) ($x $y) ()) ())))
      ($f_hyps_lists (matchc &kb (FList (FSDepth $level) $flist) ($level $flist)))
      ($f_levels (collapse (match-atom' $f_hyps_lists ($l $_) $l)))
      ($f_max_level (if (== $f_levels (())) 0 (max-atom $f_levels)))
      ($f_hyps_list (collect_lists_by_depth $f_hyps_lists 1 $f_max_level Nil))
      ($f_hyps (filter' $f_hyps_list assign_f_hyp_to_var))
      ($mand_vars' (matchc &kb (MandVar $var) $var))
      ($_2 (remove-patternc &kb (MandVar $var)))
    ) ( (DVars $dvs) (FHyps (from-list $f_hyps)) (EHyps (from-list $e_hyps_list)) (Statement
     $stmt) )))
```

Implementation Sketch: Verifier Overview (verify)

```
(= (verify $proof $conclusion)
   (let*
      (
        ($_0 (treat_normal_proof $proof))
        ($stack_expr (matchc &stack ( (Num  $n) $f) $f))
        (() (if (== () $stack_expr) (Error (assertion: $conclusion) "Empty stack at end of proof.") ()))
        (() (if (> (size-atom $stack_expr) 1) (Error ((assertion: $conclusion) (stack: $stack_expr)) "Stack has more than one entry at end of proof.") ()))
        ($stack_top (car-atom $stack_expr))
        (() (println! (Comparing: $stack_top "==" $conclusion)))
        (() (if (== $conclusion $stack_top) () (Error ((assertion: $conclusion) (stack: $stack_expr)) "Stack entry does not match proved assertion.")))
        (() (println! "Correct proof!"))
      ) ()));;($stack_top)))
```

Implementation Sketch: Verifier Overview (treat normal proof)

```
(= (treat_normal_proof $proof)
   (let*
     (
       (() (println! (Got Proof: $proof)))
       ($_0 (empty-space &stack)) ;; stack: list[Stmmt] = []
       ;; active_hypotheses = {label for frame in self.fs for labels in (frame.f_labels, frame.e_labels) for label in labels.values()}
       ($_1 (matchc &kb ((Label $label) FHyp $FSDepth $Data) (add-atom &kb (ActiveHyp $label))))
       ($_2 (matchc &kb ((Label $label) EHyp $FSDepth $Data) (add-atom &kb (ActiveHyp $label))))
       ($_3 (map-atom $proof $label (treat_step $label)))
     )
     (remove-patternc &kb (ActiveHyp $_))))
```

Implementation Sketch: Verifier Overview (treat step)

Looks up the *label*'s data and passes treatment on.

```
(= (treat_step $label)
    (let*
      (
        (() (println! (»»» treating label $label)))
        (($Type $Data) (unify &kb ((Label $label) $Type $Data)
          ($Type $Data)
          (unify &kb ((Label $label) $Type (FSDepth $level) $Data)
            ($Type $Data)
            (Error (label $label) "No statement information found for label"))))
        ($stack_len (case (matchc &stack ( (Num $n) $s ) $n) ( ( () 0 ) ( $nums (+ 1 (max-atom $nums))) )))
        (() (println! ($Type $label data: $Data)))
      )
      (let ()
        (case $Type
          (
            (FHyp (treat_hypothesis $label $Type $Data $stack_len))
            (EHyp (treat_hypothesis $label $Type $Data $stack_len))
            (Assertion (treat_assertion $label $Data $stack_len))
            (Proof (treat_assertion $label $Data $stack_len))
          ))
        (println! (stack ($label): (matchc &stack $s $s))))
    ))
```

Implementation Sketch: Verifier Overview (treat hypothesis)

If the label is active, the floating or essential hypothesis is added to &stack.

```
(= (treat_hypothesis $label $Type $Data $stack_len)
  (unify &kb (ActiveHyp $label)
    (case $Type
      ( (FHyp
          (let* (
              ($typecode (match-atom' $Data (Typecode $t) $t))
              ($var (match-atom' $Data (FVar $v) $v))
            ) (add-atom &stack ((Num $stack_len) ($typecode $var)))))
        (EHyp
          (let $stmt (match-atom' $Data (Statement $s) $s)
            (add-atom &stack ((Num $stack_len) $stmt))) ))
    (Error (label $label) "The label is the label of a nonactive hypothesis.")))
```

Implementation Sketch: Verifier Overview (treat assertion)

1. Calculates how many atoms to pop from the stack.
2. Builds the substitution space from f_hyps on the stack (if the typecodes match what the assertion needs).
3. Check that each stack entry matches the substituted e_hyps in order.
4. Checks for disjoint variable violations.
5. Applies the substitution to the assertion statement, and pushes it to the stack.

```
(= (treat_assertion $label $Data $stack_len)
   (let*
    (
     ($dvars (match-atom' $Data (DVars $dvars) $dvars ))
     ($fhyps (match-atom' $Data (FHyps $fhyps) $fhyps ))
     ($ehyps (match-atom' $Data (EHyps $ehyps) $ehyps ))
     ($statement (match-atom' $Data (Statement $statement) $statement ))
     ($lf (size-atom $fhyps))
     ($le (size-atom $ehyps))
     ($npop (+ $lf $le))
     ($sp (- $stack_len $npop))
     (() (if (< $sp 0) (Error ((label $label) (npop $npop)) "Stack underflow: proof step requires too many hypotheses") ()))
     ($_0 (change-state! &sp $sp))
     ($subst (new-space)) ; ($subst &subst)
     ($_1 (map-atom $fhyps $fhyp (add-subst $subst $fhyp)))
     ($_2 (map-atom $ehyps $ehyp (check_subst $subst $ehyp)))
     ($_3 (eval (collapse (check_dvs $subst $dvars))))
     ($_4 (matchc &stack ( (Num $n) $s ) (if (>= $n $sp) (remove-atom &stack ( (Num $n) $s )) ())))
     ($new_conclusion (let $new_conclusion (apply_subst $subst $statement) (let () (add-atom &stack ((Num $sp) $new_conclusion)) $new_conclusion)))
   ) ()));(println! (stack ($label): (matchc &stack $s $s)))))
```

Implementation Sketch: Verifier Overview (...)

- The rest is on github.

Backward Chainer

```
;; Base cases
;; Match the knowledge base
(= (bc $kb $env $_ (: $proof $theorem))
   (match $kb (: $proof $theorem) (: $proof $theorem)))
;; Match the environment
(= (bc $kb $env $_ (: $proof $theorem))
   (match' $env (: $proof $theorem) (: $proof $theorem)))

;; Recursive step
;; Unary proof application
(= (bc $kb $env (S $k) (: ($rule $arg) $theorem))
   (let* (;; Recurse on unary rule
          ((: $rule (-> $premises $theorem))
           (bc $kb $env $k (: $rule (-> $premises $theorem))))
          ;; Recurse on premise
          ((: $arg $premises)
           (bc $kb $env $k (: $arg $premises))))
     (: ($rule $arg) $theorem)))
;; Binary proof application
(= (bc $kb $env (S $k) (: ($rule $arg1 $arg2) $theorem))
   (let* (;; Recurse on binary rule
          ((: $rule (-> $premises1 $premises2 $theorem))
           (bc $kb $env $k (: $rule (-> $premises1 $premises2 $theorem))))
          ;; Recurse on premise 1
          ((: $arg1 $premises1) (bc $kb $env $k (: $arg1 $premises1)))
          ;; Recurse on premise 2
          ((: $arg2 $premises2) (bc $kb $env $k (: $arg2 $premises2))))
     (: ($rule $arg1 $arg2) $theorem)))
```

Backward Chainer Friendly Form?

```
!(bind! &md (new-space))
!(add-atom &md (: ⟨0⟩ Const))
!(add-atom &md (: ⟨+⟩ Const))
!(add-atom &md (: ⟨=⟩ Const))
!(add-atom &md (: ⟨->⟩ Const))
!(add-atom &md (: ⟨⟦⟩ Const))
!(add-atom &md (: ⟨⟧⟩ Const))
!(add-atom &md (: ⟨term⟩ Const))
!(add-atom &md (: ⟨wff⟩ Const))
!(add-atom &md (: ⟨|-⟩ Const))
!(add-atom &md (: ⟨t⟩ Var))
!(add-atom &md (: ⟨r⟩ Var))
!(add-atom &md (: ⟨s⟩ Var))
!(add-atom &md (: ⟨P⟩ Var))
!(add-atom &md (: ⟨Q⟩ Var))
!(add-atom &md (: tt (: ⟨t⟩ ⟨term⟩)))
!(add-atom &md (: tr (: ⟨r⟩ ⟨term⟩)))
!(add-atom &md (: ts (: ⟨s⟩ ⟨term⟩)))
!(add-atom &md (: wp (: ⟨P⟩ ⟨wff⟩)))
!(add-atom &md (: wq (: ⟨Q⟩ ⟨wff⟩)))
!(add-atom &md (: tze (: ⟨0⟩ ⟨term⟩)))
!(add-atom &md (: tpl (-> (: $⟨t⟩ ⟨term⟩) (: $⟨r⟩ ⟨term⟩) (: ( $⟨t⟩ ⟨+⟩ $⟨r⟩ ) ⟨term⟩)))))
!(add-atom &md (: weq (-> (: $⟨t⟩ ⟨term⟩) (: $⟨r⟩ ⟨term⟩) (: ($⟨t⟩ ⟨=⟩ $⟨r⟩) ⟨wff⟩))))
!(add-atom &md (: wim (-> (: $⟨P⟩ ⟨wff⟩) (: $⟨Q⟩ ⟨wff⟩) (: ( $⟨P⟩ ⟨->⟩ $⟨Q⟩ ) ⟨wff⟩))))
!(add-atom &md (: a1 (-> (: $⟨t⟩ ⟨term⟩) (: $⟨r⟩ ⟨term⟩) (: $⟨s⟩ ⟨term⟩) (: ( $⟨t⟩ ⟨=⟩ $⟨r⟩ ⟨->⟩ ( $⟨t⟩ ⟨=⟩ $⟨s⟩ ⟨->⟩ $⟨r⟩ ⟨=⟩ $⟨s⟩ ) ) ⟨|-⟩)))))
!(add-atom &md (: a2 (-> (: $⟨t⟩ ⟨term⟩) (: (( $⟨t⟩ ⟨+⟩ ⟨0⟩ ) ⟨=⟩ $⟨t⟩) ⟨|-⟩))))
!(add-atom &md (: mp (-> (: $⟨P⟩ ⟨wff⟩) (: $⟨Q⟩ ⟨wff⟩) (: $⟨P⟩ ⟨|-⟩) (: ( $⟨P⟩ ⟨->⟩ $⟨Q⟩ ) ⟨|-⟩) (: $⟨Q⟩ ⟨|-⟩)))))
; !(add-atom &md (: th1 (-> (: $⟨t⟩ ⟨term⟩) (: ($⟨t⟩ ⟨=⟩ $⟨t⟩) ⟨|-⟩)))))
```

Backward Chainer: Goal

Can the bc find the proof of $t = t$?

```
> !(bc &self
    Nil
    (fromNumber 10)
    $proof (: ((⟨t⟩ ⟨=⟩ ⟨t⟩) ⟨|-⟩)))
[(: ((((mp-curry (weq (tpl tt tze) tt)) (weq tt tt)) (a2 tt))
  ((((mp-curry (weq (tpl tt tze) tt)) (wim (weq (tpl tt tze) tt) (weq tt tt))) (a2 tt))
  (((a1-curry' (tpl tt tze)) tt) tt))) (: ((⟨t⟩ ⟨=⟩ ⟨t⟩) ⟨|-⟩)))]
```

Backward Chainer: Goal

Can the bc find the proof of $t = t$?

A: nope, but it can replay the proof, but Nil's version is becoming readable.

```
!(assertEqual
  (bc &kbh (fromNumber 5)
      (: (mp (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩)
             (⟨=⟩ ⟨t⟩ ⟨t⟩)
             (a2 ⟨t⟩)
             (mp (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩)
                 (⟨->⟩ (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩) (⟨=⟩ ⟨t⟩ ⟨t⟩))
                 (a2 ⟨t⟩)
                 (a1 (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩ ⟨t⟩)))
         (⟨=⟩ ⟨t⟩ ⟨t⟩)))
  (: (mp (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩)
         (⟨=⟩ ⟨t⟩ ⟨t⟩)
         (a2 ⟨t⟩)
         (mp (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩)
             (⟨->⟩ (⟨=⟩ (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩) (⟨=⟩ ⟨t⟩ ⟨t⟩))
             (a2 ⟨t⟩)
             (a1 (⟨+⟩ ⟨t⟩ ⟨0⟩) ⟨t⟩ ⟨t⟩)))
     (⟨=⟩ ⟨t⟩ ⟨t⟩)))
```

What's next?

- Probably switching from MM $\rightarrow$ MeTTa to MM0/U $\rightarrow$ MeTTa.
  - MM0 already does a lot of the work we'd need to do inference over MM.
  - (Also, disjoint variables are kinda quirky and weird.)
- *Minimal MeTTa 2* (MM2) is a low-level, efficient version of MeTTa.
- It probably makes sense to explore MM0/U $\rightarrow$ MM2.

What's next?

- Probably switching from MM $\rightarrow$ MeTTa to MM0/U $\rightarrow$ MeTTa.
  - MM0 already does a lot of the work we'd need to do inference over MM.
  - (Also, disjoint variables are kinda quirky and weird.)
- *Minimal MeTTa 2* (MM2) is a low-level, efficient version of MeTTa.
- It probably makes sense to explore MM0/U $\rightarrow$ MM2.
- ... and I'm open to feedback as to what might make sense in terms of (lazily) integrating formal verification into AGIs.