NARS implementation in MeTTA

Peter Isaev, Patrick Hammer

January 2024

1 Introduction

This document provides an overview of the basic principles behind Non-Axiomatic Reasoning System (NARS) implemented in MeTTa programming language and justifies the design decisions according to the specifics of the language. It also outlines a POC demonstrating procedure learning in MeTTa and summarizes issues and limitations observed.

2 Overall Architecture

The MeTTa implementation of NARS follows design paradigms borrowed from OpenNARS for Applications (ONA)[1] utilizing Non-Axiomatic Logic (NAL)[2], concept-centric semantic memory[3] and control mechanism, including declarative, temporal and procedural types of reasoning. The inference control framework is shown in Figure 1. The control is mainly concerned with which premises to pick and which NAL inference rules to apply on a cyclic basis to derive knowledge, trigger a decision or lead to the derivations of subgoals.

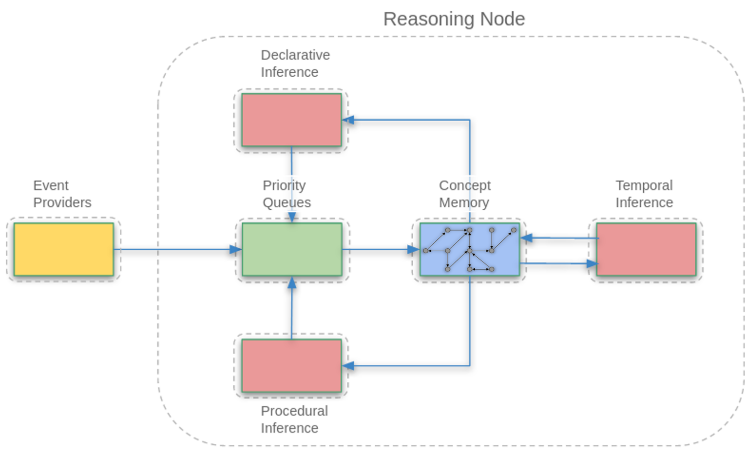

Figure 1: Overview of ONA, the NARS implementation

Describing the functionality and structure of the control components in detail is well beyond the scope of this document, however, we will recall the most relevant definitions of the components within the corresponding sections of the report to allow the reader to understand their principle functionalities.

3 Code Structure

The reasoning system consists of the following three (3) main components:

- **Logic**
- **Memory**
- **Control**

Whereby each of them is further subdivided into multiple parts as can be seen in code organization Figure 2.

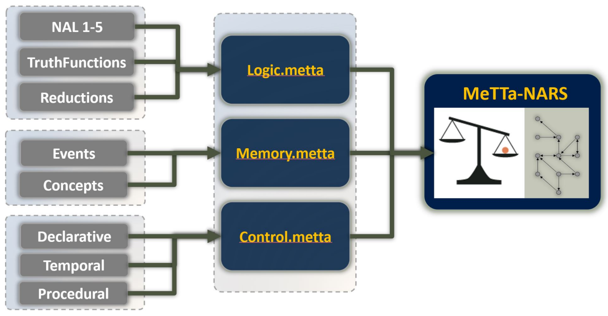

Figure 2: Code Structure Flowchart

Important to note that at the time of the creation of this document, the MeTTa language did not offer a reliable code importing process and therefore the simple concatenation of the source files was performed instead by using building scripts to construct each of the components from the corresponding smaller MeTTa files.

Hereby, the logic component is built using multiple smaller MeTTa files for NAL inference rules, NAL Truth Functions and Term Reductions, while the memory of the system is divided into Events and Concepts. Given the complexity of NARS reasoning, the control of the system is a more substantial entity consisting of the following sub-parts, each conforming to the type of reasoning:

- Declarative (deriving events and knowledge from events)
- Temporal (sequence and temporal implication formation)
- Procedural (decision making with subgoaling and planning)

The build script *build.sh* at the root of the directory performs the concatenation process of the initial source code files into 3 larger chunks and eventually into a single NARS reasoner *NARS.metta* as seen in Fig. 2.

4 Standard library extensions

In addition to the main three components discussed above, MeTTa-NARS features a small utility package within *util.metta* that defines the commonly used procedures as well as basic mathematical functions. Below are descriptions of the select procedures defined within *util.metta*.

TupleConcat:

TupleConcat concatenates two given tuples into a single one by explicitly using inbuilt *superpose*.

```
(= (TupleConcat $Ev1 $Ev2)
    (collapse (superpose ((superpose $Ev1) (superpose $Ev2)))))
```

Basic math functions:

```
(= (max $1 $2) (if (> $1 $2) $1 $2))
(= (min $1 $2) (if (< $1 $2) $1 $2))
(= (abs $x) (if (< $x 0) (- 0 $x) $x))
```

Sequential:

A wrapper for the sequential instructions evaluation and modifications. Currently using inbuilt *superpose* however might face certain concerns as noted in the "Issues and Limitations" section.

```
(: sequential (-> Expression %Undefined%))
(= (sequential $1) (superpose $1))
```

CollapseCardinality:

Given that MeTTa currently does not provide an easy inbuilt way of knowing the length of a tuple, *CollapseCardinality* is used to determine tuple size. To allow this happening, *BuildTupleCounts* creates a map by adding tuples of up-to the given size with their cardinality to the atom space in the background.

```
(= (BuildTupleCounts $TOld $C $N)
(let $T (collapse (superpose (1 (superpose $TOld))))
    (superpose ((add-atom &self (= (TupleCount $T) (+ $C 2)))
                (If (< $C $N) (BuildTupleCounts $T (+ $C 1) $N))))))
```

Do and If:

Do is trivial procedure returning an empty tuple, idea of returning *None*, when the output is not desired, thus many functions within the control code are wrapped in *Do* procedure. Additionally, since MeTTa only provide *If-else* construct, more common definition of *if-else* statement is provided.

```
(: do (-> Expression %Undefined%))
(= (do $1) (case $1 ()))
(: If (-> Bool Atom Atom))
(= (If True $then) $then)
(= (If False $then) ())
(: If (-> Bool Atom Atom Atom))
(= (If $cond $then $else) (if $cond $then $else))
```

Please note, the type annotations are necessary to hamper it from evaluating the input expression.

5 The Logic

Non-Axiomatic Logic (NAL)[2] specifies which information can be derived from combinations of at most two premises with support of uncertainty estimation. It incorporates inference rules for learning from event streams, goal reasoning and decision making, as well as functions to estimate the uncertainty of statements based on evidential support. Hence, a means-end reasoner can be built to learn from experience using NAL.

NAL is defined using multiple layers (1-5) with corresponding truth functions for uncertainty estimation, whereby logic becomes more advanced with each layer. Inference Rules along with truth functions are being explicitly defined within the Logic component of MeTTa-NARS. Below are examples of basic inference rules representation following the convention of Hyperon-PLN which uses entails operator.

NAL-1 Inference Rules for Inheritance:

```
(= (|- (($a --> $b) $T1) (($b --> $c) $T2)) (($a --> $c) (Truth_Deduction $T1 $T2)))
(= (|- (($a --> $b) $T1) (($a --> $c) $T2)) (($c --> $b) (Truth_Induction $T1 $T2)))
(= (|- (($a --> $c) $T1) (($b --> $c) $T2)) (($b --> $a) (Truth_Abduction $T1 $T2)))
```

NAL-5 Inference Rules for Implication and Equivalence:

```
(= (|- (($A ==> $B) $T1) (($B ==> $C) $T2)) (($A ==> $C) (Truth_Deduction $T1 $T2)))
(= (|- (($A ==> $B) $T1) (($A ==> $C) $T2)) (($C ==> $B) (Truth_Induction $T1 $T2)))
(= (|- (($A ==> $C) $T1) (($B ==> $C) $T2)) (($B ==> $A) (Truth_Abduction $T1 $T2)))
(= (|- (($A ==> $B) $T1) (($B ==> $C) $T2)) (($C ==> $A) (Truth_Exemplification $T1 $T2)))
(= (|- (($S <=> $P) $T)) (($P <=> $S) (Truth_StructuralIntersection $T)))
(= (|- (($S ==> $P) $T1) (($P ==> $S) $T2)) (($S <=> $P) (Truth_Intersection $T1 $T2)))
(= (|- (($P ==> $M) $T1) (($S ==> $M) $T2)) (($S <=> $P) (Truth_Comparison $T1 $T2)))
(= (|- (($M ==> $P) $T1) (($M ==> $S) $T2)) (($S <=> $P) (Truth_Comparison $T1 $T2)))
(= (|- (($M ==> $P) $T1) (($S <=> $M) $T2)) (($S ==> $P) (Truth_Analogy $T1 $T2)))
(= (|- (($P ==> $M) $T1) (($S <=> $M) $T2)) (($P ==> $S) (Truth_Analogy $T1 $T2)))
(= (|- (($M <=> $P) $T1) (($S <=> $M) $T2)) (($S <=> $P) (Truth_Resemblance $T1 $T2)))
```

Selected Truth Functions:

```
(= (Truth_Deduction ($f1 $c1) ($f2 $c2)) ((* $f1 $f2) (* (* $f1 $f2) (* $c1 $c2))))
(= (Truth_Abduction ($f1 $c1) ($f2 $c2)) ($f2 (Truth_w2c (* (* $f1 $c1) $c2))))
(= (Truth_Induction $T1 $T2) (Truth_Abduction $T2 $T1))
(= (Truth_Exemplification ($f1 $c1) ($f2 $c2)) (1.0 (Truth_w2c (* (* $f1 $f2) (* $c1 $c2)))))
```

Additionally, to allow structural term reduction during derivation process, term reduction rules are being featured in Logic component and applied at the time of deriving new or updating existing knowledge.

Selected NAL term reductions:

```
(= ($A & $A) $A)
(= ($A | $A) $A)
(= ($A && $A) $A)
(= ($A || $A) $A)
(= (({ $A }) | ({ $B })) ({ $A $B }))
(= (({ $A $B }) | ({ $C })) ({ ($A . $B) $C }))
(= (({ $C }) | ({ $A $B }) ) ({ $C ($A . $B) }))
```

6 Memory

Memory in NARS follows a concept-centric semantic memory structure in accordance with the NAL term logic the system uses. It can be viewed as a graph where concepts are represented as nodes and links designate relationships among them as in Figure 1. Technically, NARS memory is a collection of concepts representing a conceptual network with prioritized nodes. **Concept** is a major entity, an identifiable unit of system's experience that has grounded meaning. It is also considered as a unit of storage to hold various components of knowledge. For more details on functionality, memory types and semantics see [3].

In MeTTa-NARS implementation, the memory part is found within *Memory.metta* source file, which is being built using **Events** and **Concepts** located in *events.metta* and *concepts.metta* respectively. Below we provide explanations of the important procedures and data structures to allow the reader understand the functionality of the system.

Events

Events part of NARS Memory sets up two Atom spaces with related functions for Beliefs and Goals which together with Attentional Focus space, found within *concepts.metta*, serve as a working term memory of the system. In original NARS implementation the atom spaces for Beliefs and Goals are implemented using priority queues, prioritised by a priority (importance) value of items. Since at the current stage no efficient data structure is provided, we bound ourselves to use Atom Spaces and then linearly iterate over them retrieving the best candidate. Because of such limitation, the Atom Spaces have been currently limited to 10 elements. For details on budget and resource allocation of NARS, see [4], hereby we will provide an overview of procedures found within *events.metta*.

(= (ProcessBeliefEvent $Ev $t)

ProcessBeliefEvent is called from *AddBeliefEvent* found in the temporal control, which is the entry point to system. The role is to add input event to the Belief space and create or update a concept for it (discussed in the *concept* section).

(= (SelectHighestPriorityEvent $collection $t)

Currently it iteratively selects the highest priority item from either Belief or Goal spaces based on the event occurrence time and current time of the system measured in cycles (discussed in Control section).

(= (BoundEvents $collection $Threshold $Increment $TargetAmount $t)

BoundEvents maintains the constant number of items ($TargetAmount$) within the Belief or Goal Atom spaces by recursively calling itself with incremented threshold until the collection has exactly $TargetAmount$ number of items.

Additionally, the Reasoner State is defined within *events.metta*. Reasoner state is a valuable concept that captures current cycle of the system and stamp id for derivations and inputs. Reasoner state is being constantly queried by system's control during reasoning for events creation, anticipation and evidential base setups.

Concepts

The main purpose of the *concepts.metta* is to create, update and maintain system's concepts, revise beliefs and maintain attentional focus of the system. There are two Atom spaces used for *attentional focus* of the system and *concept* memory, long-term system's experience storage. At the current moment, to make the system responsive, the system attentional focus is bounded to 10 elements which suffices for running procedural learning experiments and testings, while the long-term concept memory is unbound. In addition, the *concept.metta* provides estimation of priority for the concepts. The subject of attention allocation and priority estimation will be discussed in the later section. The following select procedures are found within *concepts.metta*:

(= (RevisionAndChoice (Event $ev1) (Event $ev2)))

RevisionAndChoice revises the belief of existing knowledge with the newly added or derived knowledge given its evidential bases do not overlap, i.e. the knowledge being derived from different evidential traces. In the case of both of the beliefs share one or more components within their evidential base, the system makes the choice regarding which knowledge to utilize based on the confidence of the belief.

(= (UpdateConcept $NewEvent $t)

The given procedure creates new or updates an existing concept. In the case of the new event to the system, either an input or a derivation, the given procedure creates a concept and adds it to the *attentional focus* Atom space. If a concept for a given event is **already present**, either in *attentional focus* or in *concept* Atom spaces, the procedures calls *RevisionAndChoice* to revise beliefs, updates the concept's priority and insert the concept into *attentional focus* Atom space. It is important to note that the concept only exists within *attentional focus* or *concept* Atom spaces but not in both, thus if a concept has lost the competition for resources it remains only in the *concept* Atom space.

(= (BoundAttention $Threshold $Increment $TargetAmount $t)

BoundAttention is being called from declarative control (discussed later), the purpose is to maintain the size of the *attentional focus* Atom space. Concepts whose priority is less than the given threshold are being moved from *attentional focus* to *concepts* Atom space. The procedure recursively calls itself until the size of *attentional focus* has reached provided target number.

7 Control Mechanism

Attentional control is performed by using the Beliefs and Goals spaces as a priority queue where in each cycle only the highest-priority event is selected. Hereby input events have priority 1.

For derived belief events, the priority is the parent event's priority (the first premise), multiplied with the belief concept's priority (where the second premise came from), multiplied with the truth expectation of the conclusion event. For derived goal events, there is no belief concept involved so that factor is omitted.

```
(= (ConclusionPriority $EPrio $CPrio $ConcTV)
(* (* $EPrio $CPrio) (Truth_Expectation $ConcTV)))

(= (SubgoalPriority $EPrio $ConcTV) (* $EPrio (Truth_Expectation $ConcTV)))
```

8 Data Structures and corresponding spaces

8.1 A space as a PQ

8.1.1 Maximum element selection

The following function can be used to extract a maximum-valued element (according to a passed function which assigns an element a value) from a tuple, which can be used to find a maximum element of a tuple and hence collapse call and space via (collapse (get-atoms &space)).

```
;retrieve the best candidate (allows to use tuples / collapse results / spaces as a PQ)
(= (BestCandidate $tuple $bestCandidate $evaluateCandidateFunction $t)
  (if (== $tuple ())
      $bestCandidate
      (let* (($head (car-atom $tuple))
             ($tail (cdr-atom $tuple)))
            (if (> ($evaluateCandidateFunction $head $t)
                  ($evaluateCandidateFunction $bestCandidate $t))
                (BestCandidate $tail $head $evaluateCandidateFunction $t)
                (BestCandidate $tail $bestCandidate $evaluateCandidateFunction $t)))))
```

8.2 Bound via thresholding

In NARS priority (attention) values are between 0 and 1. Since this is the case, starting from 0 one can slowly increase a bound, removing items below that bound, until the space only contains the desired maximum amount of items.

```
;bound the size of the attentional focus for tasks / events
(= (BoundEvents $collection $Threshold $Increment $TargetAmount $t)
   (sequential ((let* (($Ev (get-atoms $collection))
                       ((Event $Sentence ($Time $Evidence $EPrio)) $Ev))
                      (if (< (EventPriorityNow $EPrio $t) $Threshold) (remove-atom $collection $Ev) nop))
                (let $CurrentAmount (CollapseCardinality (get-atoms $collection))
                     (if (and (> $CurrentAmount $TargetAmount) (< $Threshold 1.0))
                         (BoundEvents $collection (+ $Threshold $Increment) $Increment $TargetAmount $t) nop)))))
```

8.3 Belief updating

```
;;update beliefs in existing concept with the new event or create new concept to enter the new evidence
(= (UpdateConcept $NewEvent $t)
   (let* (((Event ($Term $TV) ($Time $Evidence $EPrio)) $NewEvent)
          ($NewEventEternalized (Eternalize $NewEvent))
          ($MatchConcept (Concept $Term $Belief $BeliefEvent $CPrio)))
         (sequential ((case (match &attentional_focus $MatchConcept $MatchConcept)
                            (($MatchConcept (sequential ((remove-atom &attentional_focus $MatchConcept)
                                                         (let* (($RevisedBelief
                                                                  (RevisionAndChoice $Belief $NewEventEternalized))
                                                                ($MaxPrio (if (> (EventPriorityNow $EPrio $t)
                                                                                 (ConceptPriorityNow $CPrio $t))
                                                                              $EPrio $CPrio)))
                                                         (add-atom &attentional_focus
                                                                   (Concept $Term $RevisedBelief $NewEvent $MaxPrio))))))
                             (%void% (case (match &concepts $MatchConcept $MatchConcept)
                                           (($MatchConcept (sequential ((remove-atom &concepts $MatchConcept)
                                                                        (add-atom &attentional_focus $MatchConcept)
                                                                        (UpdateConcept $NewEvent $t))))
                                            (%void% (add-atom &attentional_focus
                                                              (Concept $Term $NewEventEternalized $NewEvent $EPrio)))))))))))
```

A few important mechanisms are at play here:

1. Retrieving evidence from concept memory into attentional focus if a concept with the term of the event already exists
2. Creating of a new concept if there was no matched one in attentional focus and also not in concepts (realized by nesting case expressions)
3. Definition of a match pattern which has variables which are resolved in the lines after.
4. An implicit constraint of the term of the concept (represented by the match pattern) to have the same term as NewEvent has as the variable $Term is used again in the definition of MatchConcept.

9 Issues and workarounds

1. The first issue, discovered on August 11, had to do with inefficiency (multiple seconds for just 10 items) Recursive tuple/list performance issue when counting the amount of items in a space. This was traced down to being slow in recursive list deconstrunction (and construction). The workaround was building a lookup table via add-atom. BuildTupleCounts
2. The second issue was about import statements. Import issue 1 (which as Alexey Potapov found has to do with superpose not including stdlib type definitions) and Import issue 2 which is yet unexplained. A satisfactory workaround was to glue files together with Linux cat command, e.g.

   ```
   cat file1.metta file2.metta > RUN.metta && metta RUN.metta
   ```

10 Tests

- tests0.metta: NAL inference rule application examples
- tests1.metta: Multistep declarative inference example
- tests2.metta: Pong-like procedure learning example
- tests3.metta: Multi-step decision making (planning) example

11 Future Research and Optimizations

While manipulating large lists with even 10's of items recursively can take seconds in MeTTa at this stage, some operations on spaces are relatively fast. For instance lookup for a pattern is quick. This can be exploited for instance in evidental base overlap check by generating all the tuples up to a certain length.

Furthermore, spaces can be used as sets to check for evidential base overlaps faster than recursive versions.

References

[1] P. Hammer and T. Lofthouse, *'OpenNARS for Applications': Architecture and Control*, 07 2020, pp. 193–204.

[2] P. Wang, *Non-axiomatic logic: A model of intelligent reasoning.* World Scientific, 2013.

[3] P. Isaev and P. Hammer, "Memory system and memory types for real-time reasoning systems," in *Artificial General Intelligence*, P. Hammer, M. Alirezaie, and C. Strannegård, Eds. Cham: Springer Nature Switzerland, 2023, pp. 147–157.

[4] ——, "An attentional control mechanism for reasoning and learning," in *Artificial General Intelligence*, B. Goertzel, A. I. Panov, A. Potapov, and R. Yampolskiy, Eds. Cham: Springer International Publishing, 2020, pp. 221–230.