# High Level Guide of AS-MOSES port to MeTTa

Nil Geisweiller

March 5, 2025

**Abstract**

This document is a high level guide of the AS-MOSES port from OpenCog Classic to MeTTa. It also discusses the notion of Cognitive Synergy in the context of MOSES as well as in the broader context.

# Contents

# Chapter 1

# Introduction

As the title suggests, this document is a high level guide of the AS-MOSES port from OpenCog Classic to MeTTa. It is primarily addressed to the i-Cog team in charge of realizing that port. In fact this very document exists because Yeabesera Derese, currently the lead of the AS-MOSES port project, asked me to write it. This document is likely to attract a broader interest though. As I started to write, it rapidly became clear that it required to treat somewhat thoroughly the aspect of cognitive synergy, which actually makes the major part of the document. Please, take it as an initial proposal though, and a proposal among many. It is clear that for cognitive synergy to be realized effectively, it requires a concerted agreement across the creators (and users) of Hyperon, as well as an ongoing exploration.

Because I want to make the point clear, let me quickly summarize the main idea of cognitive synergy I present in this document. Pragmatically speaking, the main idea is to replace handwritten heuristics appearing throughout the AS-MOSES code base by explicit logically formulated queries. Heuristics exist because the optimal choices are too difficult to make. That is precisely where Hyperon comes to the rescue. When a choice is too difficult to make, instead of offering a hardwired and ultimately incomplete solution, poke a hole, look outside and ask Hyperon for help.

The outline of the document is as follows. The rest of this Chapter provides a brief history of AS-MOSES and a few words about how I conceive the goal of the port. Chapter 2 discusses aspects of the port specific to MOSES as implemented in OpenCog Classic. Chapter 3 delves into the practicalities of Cognitive Synergy, between MOSES and Hyperon, MOSES and humans, MOSES and MOSES, alongside comments on improvements and miscellaneous considerations, including the perspective of framing MOSES as a form of reasoning.

## 1.1 AS-MOSES, a brief history

MOSES [13], which stands for *Meta-Optimizing Semantic Evolution Search*, is an evolutionary program learner initially developed by Moshe Looks. It takes in input a problem description, and outputs a set of programs attempting to solve that problem. A typical example is the problem of fitting data, in that case the problem description may be a table of inputs and outputs alongside a fitness function measuring how well a candidate fits that data.

I believe the primary motivation behind the creation of MOSES came from the desire to adapt existing EDA (Estimation of Distribution Algorithm) methods to evolve programs instead of bitstrings. Upon investigating that space, Moshe, with the help of Ben Goertzel, discovered that by combining a few tricks, evolving programs using EDA was actually competitive. We will come back to these tricks but, let me say that, as with anything, these tricks tend to work well under some assumptions, and not so well under some others, which will attempt to recall as well.

At some point Moshe left, and Linas Vepstas and I took over the MOSES code base, added a number of improvements such as stochastic hillclimbing, crossover, bootstrapping, diversity management and more. We, however never got a chance to improve the EDA part (remaining stuck in an incomplete stage), which is probably regrettable because, I believe, if done well, this method has a lot to offer. I do not expand too much about improving EDA in this document, only offer a few hints. Maybe this subject deserves a future document of its own.

In terms of performances, MOSES did achieve good results for its time specifically for evolving programs restricted to Boolean expressions. It performed poorly for programs involving floating point numbers (in spite of Moshe inventing a clever representation of numbers amenable to discreet linkage learning). Later on, it was also used to evolve programs controlling agents to perform imitation learning in virtual environments, with some amount of success.

Initially, the target programming language (i.e. the representational language of the candidates being evolved) supported by MOSES was its own thing, called Combo and described by Moshe as "Lisp with a bad haircut". Later on, as OpenCog Classic developed, the need to integrate MOSES more deeply into OpenCog came to be. And the task of replacing Combo by Atomese, the language of OpenCog Classic (the equivalent of MeTTa for Hyperon) was initiated, and carried out, for the most part, by Kasim Ebrahim. The result was called AS-MOSES, for AtomSpace-MOSES. That endeavor was never completed because Kasim left, and the development effort was shifting from OpenCog Classic to Hyperon anyway.

The repositories of MOSES can be found here [7] and that of AS-MOSES can be found there [1]. There are reasonably similar. The main difference is the AS-MOSES repository contains code for evolving Atomese programs beside Combo. The AS-MOSES code base is, in some limited respects, a bit cleaner, but also a bit heavier due to the additional Atomese support. In the rest of the document I will often mention MOSES while meaning either MOSES or AS-

MOSES, or MOSES Classic to denote versions coming from OpenCog Classic as opposed to future versions of MOSES for Hyperon.

## 1.2 The Goal of the Port

I believe the goal of the port should not be to verbatimely reproduce MOSES inside Hyperon. The goal, in my opinion, should be to

*Create a sufficiently open-ended program learning framework that integrates well with the rest of Hyperon to enable some form of cognitive synergy.*

Indeed, even though MOSES contains important innovations that we want to be ported, it largely missed the cognitive synergy aspect.

Pragmatically speaking, cognitive synergy here means that if MOSES gets stuck in the process of evolving programs, it can formulate a request of help to the rest of Hyperon and take advantage of it to unstick itself. Likewise, if other components of Hyperon are stuck, they can formulate requests of help to MOSES and take advantage of it. There are certainly many ways such cognitive synergy could be realized, I will describe one that I like, or that at least I understand, but ultimately how to do that well, i.e. producing synergy as opposed to interference, remains an open field of research.

There is also another form of synergy, perhaps just as important, the synergy between MOSES and its users. This could for instance take the form of integrating MOSES to the MeTTa LSP server. More will be said on that further in the document.

# Chapter 2

# Porting MOSES Classic

## 2.1 MOSES Secret Sauce

Let me briefly recall the main tricks that contributed to make MOSES competitive.

1. Reducing candidates to normal form. That trick consists in applying rewriting rules to transform candidates into some canonical form while preserving semantics. For example, given the commutativity of conjunction, `(and y x)` could be rewritten into `(and x y)`, where its arguments have been swapped to follow a predefined order. Thus if MOSES generates both `(and x y)` and `(and y x)`, after reduction they would both point to the same candidate. This has the following advantages:

   (a) Avoid re-evaluating syntactically different, yet semantically identical candidates, saving resources in the process.

   (b) Candidates that are more consistently formatted are also easier to recombine meaningfully.

   (c) Increase syntactic versus semantics correlation. It is possible to design reduction rules so that candidates that are syntactically similar after reduction are also more likely to be semantically similar. This has the effect of making the fitness landscape less chaotic, therefore less deceptive. The Elegant Normal Form, used in the reduction engine for Boolean expressions, happens to have such property. It also happens to have the property of not utterly destroying the structure of a Boolean expression, making it more suitable for recombination. This is aligned with of one of the core assumptions underlying MOSES competitiveness, which is that *the problems we care about tend to be decomposable*.

2. Locally vectorizing the search space. Given an exemplar candidate, MOSES generates a program subspace around that candidate, called a *deme*. Such

subspace being a vector space makes it amenable to a battery of optimization techniques, including hillclimbing, cross-over and of course EDA. Upon optimizing a deme, MOSES collects promising candidates that can subsequently be used as exemplars to spawn more demes. In the code of MOSES, the process of locally vectorizing the search space is called *representation building*.

3. Vectorizing the fitness. Instead of reducing the fitness to a single number, the fitness can be represented as a vector of components, providing some support for multi-objective optimization. For example, MOSES can be asked to retain only the Pareto front of a population. Additionally, diversity pressure can be applied during the search by taking into account the distance between these vectorized scores. Typically, in MOSES components would represent the fitness of the candidate for each data point, as opposed to just its aggregated fitness. In the code of MOSES, such vectorized fitness is called *behavioral score*.

As always, it was observed that these tricks could speed-up evolution in some situations and slow it down in others. Choosing the right hyper-parameters for a given problem was a difficult task.

## 2.2 Porting MOSES's Tricks to Hyperon

What is enumerated above is not the full set of tricks MOSES used, but constitute a good starting point for porting MOSES to Hyperon. Let me provide some high-level guidance in that respect.

1. Reducing candidates to normal form. I believe this can be elegantly ported using either the native MeTTa pattern-based interpreter, or the chaining technology developed in [2]. One technical difficulty when doing so is that program variables can be confused with program holes during reduction (via evaluation or chaining). Using De Bruijn index is one way to address that problem. Then, it requires to decompose the Elegant Normal Form algorithm as an explicit set of rewriting rules. This may take effort but it is worth it because it may then offer more flexibility and extendability. Indeed, some problems require weaker or stronger forms of reduction. By breaking down a monolithic implementation into rules, one can more easily assemble subset of rules to control the reduction strength on a per situation basis. Additionally, it makes generalizing reduction beyond the Boolean domain more natural. Some rules such as reordering the arguments of a commutative operator may already go beyond the Boolean domain. Another advantage of implementing the reduction engine in that way is that it makes it easy to support new operators as long as they come with the descriptions of their algebraic properties. Finally, reframing reduction as a form of reasoning opens the door to more flexible hybridization between evolution and reasoning.

2. Locally vectorizing the search space.

   (a) Even though vectorizing the search space is a convenient and powerful way to represent a subspace to optimize, the port should also explore alternatives. It has also drawbacks, one being that what is learned in a local representation is not necessarily easy to transfer to other representations. Indeed, the wisdom accumulated while searching a deme should ideally be transferable to other demes. For instance, in MOSES Classic, EDA was taking the form of learning a Bayesian Network over the components of that vector space. But since the semantics of each component was not the same for other vector spaces, the knowledge accumulated during the optimization of a deme would not be directly transferable to other demes. I thus recommand to port that aspect but also to explore other, perhaps more global, representations (as the i-Cog team has already started).

   (b) Also, the Bayesian learning phase did not take into account the confidence of the probabilities learned, which in turn made it difficult to properly balance exploration and exploitation during the sampling phase. PLN, which has a native support for confidence, could potentially be used as a replacement. There are certainly also promising avenues to explore to combine EDA and hillclimbing. I believe the litterature already contains such research.

   (c) I should mention that when vectorizing a program subspace, there is an inherant tension between expressivity and regularity. The more expressively dense a representation is, the more chaotic and potentially deceptive it likely is as well, so vectorizing should be flexible and easily programmable. Among the set of possibilities, perhaps the following paper could be relevant [11]. Likewise, there is the problem of minimizing over-representation in that vector space, meaning that ideally one vector should correspond to one candidate. MOSES contains heuristics to minimize that form of over-representation by cleverly using, albeit heuristically as well, the reduction engine to discard dimensions at representation building time.

3. Vectorizing the fitness. I think MOSES did a good job there and it could potentially be ported as it is. There are a number of diversity distances that could be ported as well. Although to be perfectly clear, as every hyper-parameters in MOSES, choosing when and when not to use such diversity pressures was difficult. For instance, retaining only the Pareto front would speed up the search for some problems, but slow it down for some others. Same thing for diversity pressure and other diversity related hyper-parameters. Regarding fitness functions, there is something that can be pushed to the next level though, to not only decompose fitness into multiple components but to make the whole fitness a clear box, amenable to analysis and reasoning, as opposed to an opaque black box only used for candidate evaluation. See Section 3.4.1 for a discussion on that.

# Chapter 3

# Cognitive Synergy in Practice

The notion of *Cognitive Synergy* [12] was introduced a while ago by Ben Goertzel. It is broadly speaking the idea of having algorithms with different sets of strengths work in concert to overcome their weaknesses. In this Chapter I attempt to provide a pragmatic approach to realize cognitive synergy in Hyperon. None of the ideas presented here are really new but the level of details provided in the context of MOSES and Hyperon is.

## 3.1 Synergy between MOSES and Hyperon

How can MOSES be helped by Hyperon? First, hyper-parameters controlling MOSES, such as for instance the portion of evaluations that should be allocated to search any particular deme, can potentially be tuned by Hyperon. Second, a big help would probably go to the optimization phase. This is after all where most of the computational resources are spent. Selecting the right optimization algorithm for the right problem is an example (which may fall under the hyper-parameter tuning aspect depending on how it is implemented, most hyper-parameters could in fact have an `auto` setting wish would allow MOSES to query Hyperon's wisdom on the fly). If the optimization algorithm is EDA-based, an important avenue for help is in modelling the fitness landscape, which, as mentioned above could use PLN. There is also the problem of transfering knowledge across demes, and ultimately across problems as well. For instance ideally the wisdom accumulated to effectively sample a deme should not be thrown away when a new deme is created. Speaking of sampling, I want to briefly mention an idea suggested by Ben Goertzel a while ago that remains quite relevant to Cognitive Synergy and the approach suggested in that document. The idea is to frame problem solving as queries to a universal sampler, called *SampleLink* [8], concentrating in one entry point the wisdom and solving capability of the whole system. Here, we do not make commitments as to whether that solver should

be or not presented as a sampler, but the idea of funneling problem solving towards a converging point remains.[1]

So how to achieve that? The main idea suggested here would be for MOSES to query Hyperon for knowledge capturing the relationships between problem, state of the search, direction to pursue and expected outcomes. For instance in the context of hyper-parameter tuning, such query may look like

*Given problem $\pi$, find me a range for hyper-parameter $f$ that is likely to achieve good performance.*

Then given knowledge present in Hyperon, like

*If problem has property $p$ and hyper-parameter $f$ is within range $r$, MOSES is expected to achieve greater than average performance with probably $\rho$.*

The querying mechanism would then be able to use that knowledge to infer, given that $\pi$ has property $p$, that hyper-parameter $f$ should be set to a value within range $r$. The same idea would apply to finer aspects of the behavior of MOSES, such as those pertaining to optimization. In the case of EDA-based optimization, such knowledge could look like

*If deme $d$ has property $p$, then if candidate $c$ contains operator $o_1$ at location $l_1$ and operator $o_2$ at location $l_2$, it is likely to be good fit with probably $\rho$.*

The EDA procedure could, at particular stages, query the atomspace for such wisdom. If none is retreived, then it would proceed as usual, but if some is, then it would be able to take advantage of it and depart from its default behavior.

What logic should be used to formulate that knowledge is an open question. PLN is an obvious candidate, so would be some type theoretic formalism, in fact both are likely to converge at some point. How that knowledge would be acquired is a vast subject, but in essence this task would be delegated to Hyperon as a meta-learning process. When should MOSES run in auto-pilot and when should it query the wisdom of Hyperon? I suppose a rule to follow would be

*When the decision is hard to make, ask Hyperon for help.*

Let me give another example. Let us say there is a certain conditional somewhere in the code base of MOSES

```
(if C B1 B2)
```

[1] One may be tempted to draw parallels between this and Tononi's Integrated Information Theory, or more generally and philosophically speaking, the concept of self-consciousness. But as tempting as it is we will not do that here.

and `C` happens to be difficult to establish, then `C` should be formulated as a query to Hyperon. In other words, anything that is too hard for MOSES to do by itself, should be delegate to Hyperon. While MOSES is running, these queries and their results should also be recorded, because this will be used to inform Hyperon about what needs to be improved. If the query corresponding to condition `C` often comes back unanswered, or answered with low confidence, it gives a cue to Hyperon that, in order to better help MOSES in the future, it must find ways to better answer this query.

Regarding *querying Hyperon*, for starter one may simply think of regular *pattern matching query*, which, with the proper back-end, should be fairly efficient. Thus the idea would be

*If Hyperon already knows the answer, then it can help MOSES on the spot by simply finding a match answering the query. Otherwise, if it cannot not help, MOSES defers to a default behavior, but Hyperon keeps a trace of the interaction for future improvements.*

A more sophisticated notion of *querying Hyperon* can be extended by introducing chaining. In that case *pattern matching* would be replaced by *reasoning*. It means however that the effort spent in such reasoning needs to be properly controlled. That could be achieved by for instance guarantying that some temporal upper bounds are met to make meta-reasoning about the overall efficiency of MOSES easier.

Next, how can MOSES helps Hyperon? Any problem that can be formulated as solved by finding programs maximizing some fitness function could in principle be solved by MOSES. Of course MOSES will tend to perform better on some problems and worse on others. So, determining whether MOSES can help in some particular situation amounts to being able to formulate the proper fitness function and then evaluate how efficient MOSES can operate on it, which is a knowledge that Hyperon should progressively build up and use to help other Hyperon processes. As a side note, MOSES could in principle discover patterns inside its own traces. Thus by applying MOSES at the meta-level, MOSES could in fact help itself.

## 3.2 Using a Common Language

There are likely multiple ways cognitive synergy can be realized. What I am going to present is the use of a common language to communicate between parts of Hyperon. Let me reuse the old notion of *Mind Agents* from OpenCog Classic to denote such parts, cognitive processes operating within an Hyperon ecosystem. Mind agents would be for instance MOSES, ECAN, a frequent subgraph pattern miner, a chainer running PLN, etc. The idea is that all mind agents would share a common language to formulate requests of help to each others. An obvious choice of language within Hyperon is MeTTa, but there

are many ways MeTTa can be used to formulate such queries. Whether it is premature or not to make that choice, for expository purpose and to get started with something, I will settle down to a specific formalism based on Dependent Types for the rest of the document. I am not claiming this formalism is a long term solution, even though I believe it is a good choice to begin with. Again, PLN, whichever form it may take in the future, could be a longer term choice for such common language. But for now, let me explain how such dependently typed language for MeTTa can be used for cognitive synergy.

The idea would be that when calling a mind agent, the description of the problem to solve would be provided in that common language. In a way, it would not matter if the mind agent is MOSES, a chainer, or something else, the query would essentially look the same. The basic format for calling such mind agent could be

```
(MIND_AGENT HYPER_PARAMETERS QUERY)
```

where

- `MIND_AGENT` is a MeTTa function, such as `moses`, for MOSES, or `bc` for the backward chainer, etc.
- `HYPER_PARAMETERS` is a data structure containing all the hyper-parameters for the call. That structure would contain for instance various default heuristics, how much simplicity or diversity pressure to apply, various references to spaces containing relevant knowledge, etc.
- `QUERY` is the query itself, a description of the problem to solve. That description does not necessarily have to be big and complex because it can take advantage of a vocabulary defined in spaces referenced inside the `HYPER_PARAMETERS` structure.

For instance, if one wishes to evolve a program computing a binary function that fits a certain data set, one may express that with

```
(moses MOSES_HYPER_PARAMETERS
       (: $cnd_prf (Σ (-> Bool Bool Bool) (FitMyData $fitness))))
```

where

- `MOSES_HYPER_PARAMETERS` is a structure, left undefined for now, containing all hyper-parameters of MOSES, .
- `(-> Bool Bool Bool)` is the type signature of the candidate we are look for.
- `FitMyData` is a parameterized type representing a particular custom fitness measure. For the query to be understood, `FitMyData` must be defined in a space referenced inside `MOSES_HYPER_PARAMETERS`.

- `$fitness` is a MeTTa variable representing a hole in the query to be determined by MOSES for each candidate, corresponding to the actual fitness score of that candidate.
- `Σ` is the Sigma type constructor, aka dependent sum, here used to express the existence of a candidate in a constructive manner.
- `$cnd_prf` is a hole representing an inhabitant of that sigma type, which, upon answering the query, should contain both the candidate and the proof that this candidate fulfills the query.
- What it outputs is the query itself where the holes have been filled with actual values. If more than one result exists, then the output is a superposition of results. An example of output would be

```
[(: (MkΣ (\ z (s z) (not z)) PRF1) (Σ (-> Bool Bool Bool) (FitMyData 0.2))),
 (: (MkΣ (\ z (s z) (or z (s z))) PRF2) (Σ (-> Bool Bool Bool) (FitMyData 0.5))),
 (: (MkΣ (\ z (s z) (and z (s z))) PRF3) (Σ (-> Bool Bool Bool) (FitMyData 0.8)))]
```

  which is the superposition of three results, each corresponding to a candidate. The best one, appearing last, is `(\ z (s z) (and z (s z)))`, alongside a proof, here undefined and represented by `PRF3`, that the candidate has a fitness score of `0.8`. `\` represents a lambda, while `z` and `(s z)` represent De Bruijn indices. More details will be provided on that later.

In that example `FitMyData` hides the complexity of the query. More self contained definitions can also be provided by a structured type instead of a mere symbol referring to a predefined type. How exactly that structured type would look like is beyond the scope of that document and does not matter much. All that matters is that the resulting type follows the type signature required by `Σ`, which, in that specific example, would be

```
(-> (-> Bool Bool Bool) Type)
```

meaning that `(FitMyData FITNESS)`, or whatever equivalent structured type, must describe a type constructor that takes a binary boolean function and returns a type. This is a common way to represent predicates in Dependently Typed Languages.

Maybe the user realizes that MOSES is in fact inadequate to discover such candidate and decides to try the backward chainer instead

```
(bc BC_HYPER_PARAMETERS
    (: $cnd_prf (Σ (-> Bool Bool Bool) (FitMyData $fitness))))
```

As you can see the call is almost the same, only the function being called and its hyper-parameters are different, the query is identical, because it carries the meaning of the problem. Another example would be if the user wants to discover patterns over a corpus of subtrees. In that case using the pattern miner will be far for effective that using MOSES, but the query to express the problem will

be the same in both cases.

The use of such common language does not stop here, mind agents can use this querying format internally. Let's say for instance that MOSES, within the course of its execution, has an important decision to make, having for instance to decide whether to search a given deme more deeply or abandon that deme and create a new one. A rough sketch of the code may look like:

```
(= (moses $hps $query)
   BODY ...
        (if (continue-search-deme $deme)
            SEARCH_DEME
            CREATE_NEW_DEME)
        ...)
```

The idea is that instead of having `continue-search-deme` make that decision on its own, it can formulate a corresponding question to Hyperon. If Hyperon knows the answer, MOSES can take advantage of this knowledge, otherwise it defers to a default behavior. What it means is that the code of `continue-search-deme`, instead of solely consisting of hardwired heuristics, may contain queries such as

```
(bc BC_HYPER_PARAMETERS QUERY_ABOUT_DEME_CONTINUATION)
```

Here the backward chainer is used as example because it is assumed to be somewhat universal. Maybe we want to have an even more universal access point such as `hyperon`, which is quite reminiscent to the *SampleLink* idea. Regardless, typically such queries will be parameterized to have low costs, as to not slow down MOSES in its progress. For instance the depth of reasoning used to answer could be practically null, keeping the reasoning shallow and inexpensive. That way, if Hyperon knows the answer it can help MOSES right away, otherwise, a record of its inability to answer can be kept and used as feedback to incentivize Hyperon to learn to how to help in the future.

One could envision a scenario where MOSES is being called on a series of problems, while in the background other mind agents operate some forms of meta-learning to build-up the knowledge to eventually help MOSES. This can be illustrated as follows:

```
(moses HP1 QUERY1)  |  Meta-learning ...          ; <- Fails to help
(moses HP2 QUERY2)  |  Meta-learning ...          ; <- Fails to help
(moses HP3 QUERY3)  |  Meta-learning ...          ; <- Fails to help
(moses HP4 QUERY4)  |  Meta-learning (discovery) ; <- Fails to help
(moses HP5 QUERY5)  |  Meta-learning ...          ; <- Hyperon succeeds!
```

During the first four runs, Hyperon fails to help MOSES at a particular decision point (such as deme continuation). During the fourth run the knowledge to help is finally discovered, leading Hyperon to guide MOSES during the fifth run by making it take the right branch, the one with a higher probability of success.

## 3.3 Program Evolution as a Form of Reasoning

What if querying Hyperon for help was not limited to a number of decision points throughout the program but instead was the program entirely? This is the question we will explore in this Section. To be clear, what I am suggesting here is essentially to reframe MOSES as an explicit form of reasoning. Before I begin, let me say that I am not necessarily advocating for this approach, even though I like it and is probably the approach I would take if I were to do the port myself. I also recognize that it has drawbacks, mainly

1. the unfront cost of formulating evolutionary programming as a form of reasoning;

2. the run-time cost of emulating evolution as a form of reasoning.

The main benefit in my view, is that it enables, at least in potential, the deepest levels of cognitive synergy one can hope to achieve. Besides, over time the run-time cost can be mitigated by *schematizing* (as Ben Goertzel likes to say), or by *specializing* (as Alexey Potapov likes to say) the parts of MOSES that do not require synergy with the rest of Hyperon.

Perhaps a hybrid approach can be considered from the get go, where MOSES is partly implemented as an explicit form of reasoning, and partly implemented in a more traditional functional way. The non-determinism of MeTTa may actually make these distinctions somewhat blurry.

The general idea of framing evoluation as a form of reasoning is that the problem of finding good program candidates is directly formulated as a query in logic, then MOSES merely provides an efficient inference control mechanism to fulfill such query, and by that discover such program candidates.

There are at least two ways to formulate queries, constructively and non-constructively. In the previous sections the constructive way was used, and that Section will be no different, but let me say a few words about their differences nonetheless. Perhaps the most important one, as it relates to this document, is that proving an existential statement, such as $\exists x P(x)$, in constructive mathematics, guaranties that there is a way to construct an object $a$ such that $P(a)$ holds. While in non-constructive mathematics one could indeed prove the existence of such object without ever having to be able to construct it. Pragmatically speaking, the way queries will be constructed will be different in both cases.

### 3.3.1 Constructive Way

In the constructive way, a statement leading to finding a program candidate may merely look like

$$\exists x \ \texttt{GoodFit}(x)$$

Then finding a proof of such statement is guarantied to provide a program candidate. To find more candidates one may simply find more proofs.

### 3.3.2 Non-constructive Way

In the non-constructive way, it is also possible to use reasoning to find program candidates, but the candidate must be represent as a hole inside the statement instead of being existentially quantified. And the chaining engine must be able to fill that hole during reasoning. At the end of the reasoning process, the candidate may be absent from the proof, but present inside an instantiated version of the statement. So instead of trying to prove

$$\exists x \ \texttt{GoodFit}(x)$$

which again may not necessarily lead to instantiating a candidate, one may instead attempt to prove

```
GoodFit($x)
```

resulting in a program candidate filling the hole

```
GoodFit(PROGRAM_CANDIDATE)
```

The backward and forward chainers of Hyperon [2] both support constructive and non-constructive mathematics. Even though I have a personal preference for the constructive way, I don't think the non-constructive way should be ruled out either. For reference, in OpenCog Classic, reasoning was always done in a non-constructive way. And in a way, that is still what we do in Hyperon at the level of the typing relationship. Regardless, throughout this document, we focus primarily on the constructive way.

### 3.3.3 Dependent Types

Perhaps contrary to popular beliefs, dependent types [3] can be used both to formalize constructive and non-constructive mathematics, although they particularly shine with the former. Below we will attempt to deepen our understand of how to use dependent sums to represent existentially quantified statements and to discover program candidates via reasoning.

#### GoodFit as Type Predicate

The standard way to represent existential quantification with dependent types is with Sigma types. The corresponding type constructor, $\Sigma$, can be defined in MeTTa as follows

```
(: Σ (-> (: $a Type) (-> $a Type) Type))
```

representing a type parameterized by

- a type `(: $a Type)`,
- and a type predicate `(-> $a Type)` over that type.

The first argument, a type, can be viewed as the domain over which an implicit existentially quantified variable would run. The second argument, a type predicate, can be viewed as the proposition that must existentially hold. For instance

```
(Σ Nat Prime)
```

would represent the statement that there exists a natural number that holds the property of being prime, formally

$$\exists x \in \mathbb{N},\ \texttt{Prime}(x)$$

Next, we can define the constructor by which a proof can reach `Σ`

```
(: MkΣ (-> (: $p (-> $a Type))
           (: $x $a)
           (: $prf ($p $x))
           (Σ $a $p)))
```

where

- `MkΣ` is the dependent sum data constructor, the building block to construct a proof of an existential quantification,
- `$p` is a predicate type,
- `$a` is the domain of `$p`,
- `($p $x)` is `$p` applied to an element `$x` of `$a`,
- `$prf` is a proof of `($p $x)`,
- `(Σ $a $p)` is the dependent sum expressing that there exists an element of `$a` which satisfies property `$p`.

For instance, the typing relationship between a proof that 3 is a prime number and the proposition of existence of a prime number, may look like

```
(: (MkΣ Prime (S (S (S Z))) PROOF_PRIME_3) (Σ Nat Prime))
```

where

- `(S (S (S Z)))` is the natural number 3,
- `PROOF_PRIME_3` is a proof, left undefined, of `(Prime (S (S (S Z))))`.

One may notice that, in that typing relationship above, `Prime` appears twice, once as the first argument of `MkΣ` and a second time as the second argument of `Σ`. That representation is borrowed from Idris [10], which supports implicit arguments, allowing it not to externally display such redundancy, even though it is still there internally. MeTTa can probably be made to support or emulate implicit arguments, but for now we will have to accept this redundancy. I may

sometimes remove the first argument of `MkΣ` in this document for the sake of conciseness.

Coming back to evolutionary programming, the domain would be programs with a certain type signature, and the property would be the goodness of fit. However, because we want to be able to quantify that fitness, the predicate must be parameterized by the fitness score. In the end, the Sigma type we want may look like

```
(Σ (-> Bool Bool Bool) (Fit 0.7))
```

where `(-> Bool Bool Bool)` represents the domain of the program candidates, in this case the class of binary Boolean functions, and `(Fit 0.7)` represents the predicate type expressing the class of candidates that are fit with degree 0.7. Thus, `GoodFit` has been replaced by the parameterized type `(Fit FITNESS)`. A proof for such type may look like

```
(MkΣ (Fit 0.7) (\ $x $y (and $x $y)) PROOF_OF_FITNESS)
```

where `\` represents a lambda abstraction. Note that, as of the time of writing this document, `\` is not part of the MeTTa syntax. However, JeTTa, a MeTTa compiler for the JVM, seems to already support it (though I don't know the exact syntax). Regardless, in that somewhat made-up and prospective syntax of lambda

```
(\ $x $y (and $x $y))
```

would be typed `(-> Bool Bool Bool)`, while

```
(\ $x (\ $y (and $x $y)))
```

the curried version, would be typed `(-> Bool (-> Bool Bool))`. One can verify that each argument of `MkΣ` is an inhabitant of its corresponding type, meaning

- `(: (Fit 0.7) (-> (-> Bool Bool Bool) Type))`
- `(: (\ $x $y (and $x $y)) (-> Bool Bool Bool))`
- `(: PROOF_OF_FITNESS ((Fit 0.7) (\ $x $y (and $x $y))))`

`PROOF_OF_FITNESS` is left undefined but could be not that far from an actual definition in a pragmatic use (we will look into it as we inject computations in that reasoning process to speed it up).

You may notice that a proof of `(-> Bool Bool Bool)` is in fact a program. Beautifully, the reasoning process to build that program is not different in nature from the one to build a proof of fitness for that program. In other words, by framing evolution as reasoning, synthesizing programs and searching the space

of programs become one unified process.

A prototype of the approach presented here can be found in [4]. Beware though that a few of things have been changed.

1. In order to quote programs, to avoid spontaneous reduction by the MeTTa interpreter, such as `(and True False)` $\xrightarrow{reduce}$ `False`, the vocabulary does not include MeTTa built-ins such as `and`, `True`, etc. Instead new symbols are introduced using the Unicode character set, so that `and` in MeTTa becomes **and** (in bold Unicode font) in the reasoning process, etc. This can probably be avoided by a clever use of the `quote` built-in, by implementing the chainer in Minimal MeTTa, or by other means. But for the sake of simplicity the prototype uses this Unicode trick instead.

2. Similarily, to avoid spontaneous unification from taking placed during reasoning, variables inside programs are replaced by De Bruijn indices. So for instance

   ```
   (\ $x $y (and $x $y)))
   ```

   becomes

   ```
   (\ z (s z) (and z (s z))))
   ```

   where `z` and `(s z)` are respectively the first and the second De Bruijn indices[2].

3. Actually, to avoid exacerbating combinatorial explosion, lambda abstraction is completely eliminated. So instead of introducing program variables (i.e. De Bruijn indices) via lambda abstraction, these are added to the environment at the beginning of reasoning. As a result, instead of evolving a program with the type signature

   ```
   (-> Bool Bool Bool)
   ```

   the following assumptions `(: z Bool)` and `(: (s z) Bool)` representing the Boolean types of the first two arguments of the program to evolve, are added to the environment. And thus the type signature of the program to evolve is replaced by `Bool`. Note that this gymnastic is actually exactly what the chainer does when encountering a lambda abstraction, so it is a rather natural trick. The difference is that it is done before reasoning and never during reasoning, because, as I said, introducing lambda abstraction on the fly during reasoning exacerbates combinatorial explosion. Indeed,

[2]Note that traditionally, De Bruijn indices do not need to be mentioned after a lambda, so for instance $\lambda x.x$ merely becomes $\lambda 0$. But in MeTTa, where currying is not automatic, there is a need to represent a lambda over multiple variables, which the suggested syntax does. I must say that in practice I have found that keeping De Bruijn indices after the lambda tends to enhance readability anyway.

every new lambda abstraction increases the possibilities of function applications, and every new function application provides one more body for lambda abstraction. This situation becomes rapidly unmanageable, thus why we try to avoid it. Besides, in this prototype we expect program evolution to use traditional logical connectors, `and`, `or` and `not` as building blocks, not lambda abstraction. It may be that at some point reasoning becomes so efficient that lambda abstraction can be re-introduced though, which brings us to the next section.

### 3.3.4 Reasoning Efficiently

Of course framing learning as a form of reasoning, as elegant as it may be, is only worth doing if it results in an efficient system. There are at least two ways this can be accomplished:

1. Injecting regular computation in the reasoning process. The idea is to outsource some proof obligations to some external computational processes that we trust. For instance, let's say we want to prove that $2 + 3 = 5$. One way to do this is to use reasoning exclusively, progressively transforming

   - $2 + 3$
   - to $1 + 4$,
   - then to $0 + 5$,
   - then finally to $5$,

   all that by manipulating the laws of equality and addition. Even though this way is perfectly correct it is also costly. Another way to do this is to query the CPU with an ADD instruction with 2 and 3 as arguments, get the results, and trust that the resulting equation is indeed correct. This is how `PROOF_OF_FITNESS` could be obtained in our evolutionary programming example. In fact in the prototype referenced earlier such proof is labeled `CPU` to convey the idea that "it is true because the CPU said so". Of course, in reality, more that the CPU has to say so, the function that is part of that external computational process has to be properly implemented and trusted, but we call it `CPU` to convey this idea.

2. Leveraging inference control. This is where the most speed-up can be obtained, but this is also the most difficult way. It is one of those "AGI-complete" problems, one that if you achieve it, you likely achieve AGI. A potential solution for this problem is almost the same as the one presented for cognitive synergy. A record of past inferences is stored and analyzed to extract predictive patterns used to guide an inference control mechanism for subsequent inferences. This would be the most generic solution. But one can also consider more specific handcrafted solutions, by implementing good old heuristics. Indeed, one can see a particular Genetic Programming algorithm as implementing such heuristics. There are then two ways to do that

(a) Substitute the chaining by a specific searching algorithm. In that case the only remnant of a notion of reasoning is in the way the problem and the solutions are presented (using a logical language as explained above). The advantage is that by re-using existing search algorithms, one can achieve speed quickly. The inconvenient is that it is somewhat inflexible to improvements, unless one is willing to learn search algorithms (which MOSES could in fact potentially do).

(b) Reframe the heuristics as a set of procedural rules guiding an explicit inference control mechanism. For an example of how to do that, see the following experiment [5] (beware that it is early stage). The inconvenience of that approach is that one needs to think how to encode the heuristics in such rule-based format. The advantage is that it can then easily be combined with more generic meta-learning activities. What that means is that the developer may initially provide a search heuristic, and then let Hyperon improve that heuristic on its own.

## 3.4 Other Improvements and Considerations

### 3.4.1 Black Box vs Clear Box

Regardless of whether MOSES is implemented or not as an explicit form of reasoning, the fitness function can be provided either

- as a *black box*, that can score candidate solutions and nothing more (think of a foreign function implemented in C++),
- or as a *clear box*, that can score candidate solutions and is also amenable to further analysis.

The former has the advantage of being fast, but the latter has the advantage of being flexible. Given such transparency, one could for instance invoke a reasoner to come up with a fitness estimator that is less accurate but more efficient than the original fitness function, while guarantying properties such that, for instance, the estimator dominates pointwise the original fitness so that it will never under-evaluate good candidates.

As I mentioned earlier, MOSES was able to handle multi-objective fitness, but it was in fact the only form of transparency it had with respect to the fitness function, which was implemented in C++. Using MeTTa to describe such fitness function would allow us to go much further.

### 3.4.2 Modularity

It might be useful to view MOSES itself as a collection of mind agents working in concert, communicating in a common language to achieve some form of Cognitive Synergy.

Let us provide an example for the reduction engine, using the same dependently typed language format suggested throughout the document. A call to the reduction engine may look like

```
(reduce REDUCE_HYPER_PARAMETERS
        (: $enf_prf (Σ (-> Bool Bool Bool)
                       (ENF (\ z (s z) (and (s z) z))))))
```

where

- `reduce` is the entry point of the reduction engine.
- `REDUCE_HYPER_PARAMETERS` is the set of hyper parameters for `reduce`.
- `(-> Bool Bool Bool)` is the type signature of the program to reduce in normal form.
- `ENF` is a parameterized type representing a binary predicate, with type signature

  ```
  (-> (-> Bool Bool Bool) (-> (-> Bool Bool Bool) Type))
  ```

  The first argument is a boolean binary function, the candidate not yet normalized, and the second candidate is the same candidate in Elegant Normal Form. Note that the predicate is curried in order to play nicely with the definition of `Σ`.
- `$enf_prf` is an inhabitant of the sigma type, which upon running that query would be replaced by a dependent pair with the program in Elegant Normal Form as first argument, which would be

  ```
  (\ z (s z) (and z (s z)))
  ```

  in that example (the arguments in the conjunction have merely been swapped to follow a predefined order), alongside the proof that it is indeed in Elegant Normal Form. Like in the prototype of evolutionary programming presented in Section 3.3, the proof does not have to be detailed and can be obfuscated behind a process that we trust. Likewise, the `ENF` predicate can either be provided as a built-in, with likely some rules and axioms about it contained in some auxiliary knowledge base, or provided as a structured type capturing what it means to be in Elegant Normal Form.

### 3.4.3 Internal Representation

The same format could also be used to represent data internally, which opens an additional avenue to enact cognitive synergy. Indeed, having internal representations following a common language allows other mind agents to interact with that data. For example MOSES may represent its populations of candidates as a collection of statements corresponding to the type relationships expressed in the query format

```
(: (MkΣ (λ z (s z) (not z)) PRF1) (Σ (-> Bool Bool Bool) (FitMyData 0.2)))
(: (MkΣ (λ z (s z) (or z (s z))) PRF2) (Σ (-> Bool Bool Bool) (FitMyData 0.5)))
(: (MkΣ (λ z (s z) (and z (s z))) PRF3) (Σ (-> Bool Bool Bool) (FitMyData 0.8)))
```

which may then be analyzed by another mind agent such as a frequent pattern miner to discover patterns relating candidate and fitness. If that format may look overly verbose for the purpose of internal storage, keep in mind that it might end-up being automatically compressed by the MeTTa backend (I believe MORK is able to do that for instance). And if not, one could always realize such compression explicitely by contextualizing the representation, as demonstrated in a recent chaining experiment on Modal Logic [6].

### 3.4.4 Going Beyond MOSES Classic

Since MOSES was invented, the field of evolutionary programming has advanced and we want to incorporate these advances into MOSES, as the i-Cog team has started to explore. Likewise, the existing components of MOSES can be improved, such as balancing exploration and exploitation in EDA. And of course more improvements will need to take place to help MOSES move beyond Boolean expressions.

## 3.5 Synergy between MOSES and its Users

### 3.5.1 MOSES as a Programmer Assistant

The advances of LLMs have brought advances in programming assistance. These however have deficiencies likely tied to the lack of use of formal methods to produce correct programs. Before LLMs, programming assistance technologies existed but were reserved to the obscure world of Automated and Interactive Theorem Provers (ATP and IPT). I believe MeTTa and Hyperon have the potential to bring these two approaches into a coherent and efficient whole. It is not clear to me how this is to be done properly but effort within the SingularityNET Foundation is taking place to move towards that vision. One would cite for instance the Semantic Parsing project, MeTTa-Motto, NARSE-GPT, the SingularityNET Assistant, and probably more.

On the ATP/ITP side, the use of the Language Server Protocol (LSP) seems to have become a standard. The MeTTaLog team is actually developing an LSP server for MeTTa. So one could envision to integrate MOSES, as well as some of its components such as the Reduction Engine, to the LSP server. Or maybe Tabby [9], a FOSS alternative to Copilot, could be used. Anyway, without expending on how to unify the formal side, offered by LSP, with the informal side, offered by LLMs, let me just present a scenario. A MeTTa programmer wants to implement a function but has fragmented knowledge about it, maybe has partial knowledge about its inputs and outputs, as well as partial knowledge about its properties. The programmer could enter a prompt containing examples of that functions, as well as expected properties, either in natural or formal language. If provided in natural language the programmer assistant would come

up with a formal specification. Then, after letting the programmer double check its correctness, it would produce a query for MOSES, or whatever tool is appropriate for the job, send that query to the LSP server, and wait for it to return the solutions to that query. If that process is too resource demanding to run locally, the ASI Alliance could offer such services remotely.

### 3.5.2 User as a MOSES Assistant

The other direction, human programmers helping MOSES, should be possible as well. After launching MOSES, the user should be able to interrupt it, query its state, inspect and even modify the content of its memory. In other words, the user should be able to interact and guide MOSES as it attempts to solve problems. For instance, one could place breakpoints at various locations in its code, so that when MOSES reaches these breakpoints, the user is given a change to overwrite the decision that MOSES would have otherwise taken. This sort of interactive capabilities would be useful in Hyperon in general, so maybe is to be tackled at a deeper level within Hyperon rather than MOSES. Also, it brings the question of whether MeTTa and the use of Dependent Types is a good fit for such level of interactive control. It may be that a process algebra such as the MeTTa-calculus alongside Spatial and Behavioral Types that Greg Meredith and his team are working on would be more adequate, but that is another topic for another document.

# Bibliography


[1] AS-MOSES GitHub Repository, https://github.com/opencog/asmoses

[2] Chaining GitHub Repository, https://github.com/trueagi-io/chaining

[3] Dependent Types Wikipedia Page, https://en.wikipedia.org/wiki/Dependent_type

[4] Example of Evolutionary Programming Framed as Reasoning, https://github.com/trueagi-io/chaining/tree/main/experimental/evolutionary-programming

[5] Inference Control in MeTTa, https://github.com/trueagi-io/chaining/tree/main/experimental/inference-control

[6] Modal Logic in MeTTa, https://github.com/trueagi-io/chaining/tree/main/experimental/modal-logic

[7] MOSES GitHub Repository, https://github.com/opencog/moses

[8] SampleLink OpenCog Wiki, https://wiki.opencog.org/w/OpenCoggy_Probabilistic_Programming#SAMPLE_LINK

[9] Tabby, Tabby Homepage, https://www.tabbyml.com/

[10] Idris, Idris Homepage (2021), https://www.idris-lang.org/

[11] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290353, https://doi.org/10.1145/3290353

[12] Goertzel, B.: Toward a formal model of cognitive synergy (2017), https://arxiv.org/abs/1703.04361

[13] Looks, M., Sc, B., Missouri, S.L., Louis, S.: Abstract Competent Program Evolution by Moshe Looks (2006)