Toward a New Improved OpenCog – Preliminary Design Ideas

Ben Goertzel, Alexey Potapov, Andre' Senna, SingularityNET-OpenCog Team *

July 8, 2020

Abstract

OpenCog is an open source software framework oriented toward AGI. It was formally launched in 2008, initially by open sourcing code written as back as far as 2001. A key initial motive was to provide a software platform for exploring a specific (integrative, cross-paradigm) approach to AGI formulated by project co-founder Ben Goertzel; however, the software system has since been used to explore a variety of other AGI ideas as well, and it has also been used as the foundation of several commercial software services (e.g. in bioinformatics, financial prediction and natural language processing).

A group of us involved with the OpenCog (proto-)AGI software system are now considering a the creation of a new version, **OpenCog Hyperon** – intended as a radical overhaul and upgrade of the system, potentially (but not necessarily) a from-the-ground rewrite. This brief document explains some of our high level thinking regarding OpenCog Hyperon, and calls for participation from the AGI community in thinking through the numerous tricky issues involved in designing and implementing a next-generation framework for AGI development.

Contents

1 Background and Objectives **2**
- 1.1 Preliminary Meta-comments 3
- 1.2 Toward OpenCog Hyperon 3

2 OpenCog Hyperon – Tentative Architecture Sketch **4**

3 The Concept of an "Atomese 2" Language **6**
- 3.1 Current "Atomese" 7
- 3.2 Onward to Atomese 2 8

4 Some Considerations Regarding Pattern Matching in Atomese **9**
- 4.1 Decomposing the Pattern Matching and Rule System Execution Process 10

*This document summarizes some aspects of discussions pursued during 2019-2020 by those members of the SingularityNET AI team involved with the OpenCog project

5 A Two-Layer Design for Atomese 2 **11**
5.1 Layer 1: A Generic Atomese Core 11
5.1.1 Gradual Typing 11
5.2 Layer 2: Initial Type Systems Atop the Core 11
5.3 Some Specific Type Systems of Apparent AGI Relevance 12
5.3.1 Dependent Types 12
5.3.2 IsoType Systems 12
5.3.3 Linear Types 13
5.3.4 Probability/Logic Interoperation 13
5.4 How Deeply to Bake In Tensors? 14

6 Distributed Atomspace **15**
6.1 Key Concepts 15
6.2 Provisional Assumptions and Constraints 16
6.3 Tentative Requirements 17
6.4 Handling Inconsistencies 20
6.5 Potential Implementation Directions 21

1 Background and Objectives

The original goal of the OpenCog system, when it was launched in 2008, was to serve as a software framework within which AGI with capabilities at the human level and ultimately (in some ways at least) beyond could be created.

The basic design idea was to start with the Atomspace – a dynamic weighted, labeled hypergraph knowledge store – and then create a number of "AI learning and reasoning agents" that would concurrently and cooperatively read from and write to the Atomspace. The various AI agents must cooperate and help each other rather than counteract, stymie or confuse each other – a theoretical concept formalized as "cognitive synergy".

The book *Engineering General Intelligence* [GPG13a] [GPG13b] presents in-depth ideas regarding a particular set of AI agents designed for deployment together within the OpenCog framework. These include a probabilistic logic system, an evolutionary program learning system, an attractor neural net dynamic for attention allocation, and more. The OpenCog software framework was designed to support flexible experimentation with these particular AI agents, but also with any others that might arise in the course of ongoing research and development.

In the course of experimentation with various proto-AGI and narrow-AI algorithms and projects using the OpenCog codebase, a great deal has been learned. And concurrently, of course, the AI field in general has advanced tremendously since 2008. Based on all this new learning and understanding, it now seems time to revisit the core design of the OpenCog platform, and seriously consider a major redesign and reimplementation.

A huge amount of work has gone into the current OpenCog system, so the proposal of potentially massively overhauling or utterly replacing core components of the system is not being made lightly. However, the goal of creating AGI is a huge one, and it won't

be surprising if it turns out that today, in 2020, a basic-level overhaul to the framework we've been incrementally evolving since 2008 is needed in order to get to a system that will be tractable for researchers and developers to use to create human-level AGI.

1.1 Preliminary Meta-comments

Please be aware that the ideas presented in this document are very much a work in progress, in the process of emerging from a group discussion process including many people (the explicit author list plus Nil Geisweiller, Vitaly Bogdanov, Cassio Pennachin, Sergei Rodionov, Anton Kolonin ... and the list goes on ...) [1].

This document is a bit of a pastiche. Some sections have been lifted, with modifications, from the paper "What Kind of Programming Language Best Suits Integrative AGI?" [Goe20b] that Ben Goertzel presented at the AGI-20 conference. The section on distributed processing contains some edited cut-and-pastes from larger documents by Alexey Potapov and Andre' Senna. Other parts here and there, e.g. the discussion of pattern matching and graph databases, were pasted with modifications from notes written by Alexey Potapov. In all cases judicious edits to the raw materials were made for coherence with the overall document.

A fairly important note on terminology is that we have typically referred to Atomspace as a "hypergraph" or a "generalized hypergraph" – where the generalization is that Atomspace can have links pointing to links instead of nodes, and can have links pointing to whole sub-Atomspaces. The term "metagraph" has recently been used to refer to these more generalized sorts of hypergraphs, and while it's a shift from our previous terminology, it is shorter and more elegant than "generalized hypergraph", so we will shift to the "metagraph" terminology here.

The hyperon is one among the zoo of elementary particles proposed in modern particle physics; in the poetic spirit of the use of the term "Atom" for OpenCog's core knowledge element, the nomenclature "OpenCog Hyperon" is intended as the first in a series of OpenCog versions named after different elementary particles. [2]

1.2 Toward OpenCog Hyperon

"OpenCog Hyperon" as we are conceiving it now will follow the same conceptual architecture as the current and original OpenCog system – an Atomspace plus a collection of cooperating AI agents reading to and writing from the Atomspace in a cognitively-synergetic manner. However we are seriously considering to radically revise or replace certain things that are relatively core to the current OpenCog codebase.

Three key points along these lines are:

- a key tool in OpenCog today is the Pattern Matcher, which carries out a variety of program-execution and inference operations as well as simple static pattern

[1] We must also emphatically acknowledge Linas Vepstas here; while he has not been involved in most of the recent discussion on OpenCog Hyperon that led up to this document, his work on OpenCog and various discussions with him over the years have informed the thinking represented here in countless ways.

[2] Some previous discussions used the term "OpenCog 2" but this was always somewhat tongue in cheek since the existing version of OpenCog is nowhere near 1.0, though some components are perhaps at that level of maturity.

matching. There is a URE (Unified Rule Engine) running on top of the Pattern Matcher, which is in turn used as the basis for implementing AI tools like the PLN (Probabilistic Logic Networks) inference engine and the Pattern Miner. What the Pattern Matcher and URE do is valuable and necessary, but it may be that the functionalities they embody should be organized in a quite different way.

- OpenCog's Atomspace currently is mainly used within the RAM of a single machine. There is a Postgres backing store which enables use of multiple single-machine Atomspaces in a hub-and-spokes architecture. This is very effective in some settings, but is far from the massive scalability across multiple distributed machines and persistent stores that is achieved by some contemporary graph databases.

- Some of our recent research has involved interfacing OpenCog with external deep neural net libraries, in order to achieve neural-symbolic functionalities (with OpenCog contributing the symbolic portion). The current OpenCog implementation enables this interfacing to happen, but it makes it awkward in some ways – this is not what the Pattern Matcher was designed for, and if this sort of interfacing with external neural net processes (plus other external frameworks and processes used to ground Atoms of various sorts)is taken as a basic requirement, then one can think of different ways to orchestrate various operations than what OpenCog currently does.

2 OpenCog Hyperon – Tentative Architecture Sketch

Figure 1 summarizes the tentative "OpenCog Hyperon" architecture that has been the subject of recent discussions. Key aspects here are:

- A distributed Atomspace (aka metagraph or weighted, labeled generalized hypergraph knowledge store) that operates across a large number of machines

- A "local cache" Atomspace hosting the portion of an Atomspace on a single machine (this is what the current Atomspace code handles, in its own way)

- Middleware for coordinating various local cache Atomspaces and the distributed Atomspace

- Code for doing vector embeddings and symbolic pattern mining from the distributed and local Atomspaces

- An "Atomese" language for representing multiple forms of knowledge in the Atomspace, with "sugared" human friendly version and a more efficient Atomspace-internal version

- Various AI agents operating directly on local Atomspaces, performing various sorts of learning, reasoning, creative hypothesis formation, and so forth

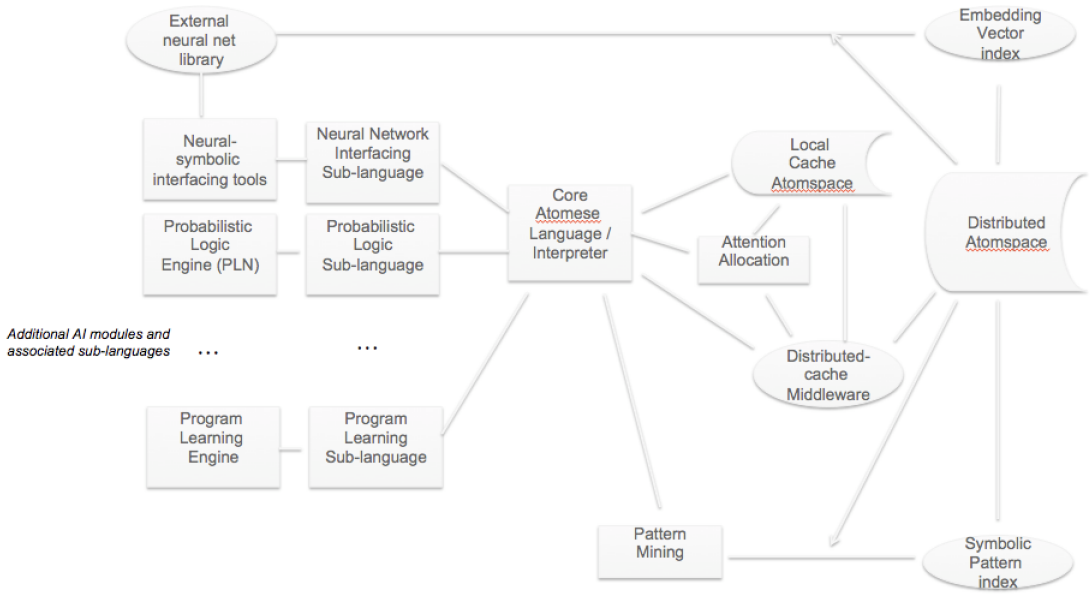

Figure 1: A software architecture for integrative AGI, with a pattern-matching-focused, gradually typed Atomese language at the core.

- Attention allocation processes that identify which Atoms should be pushed into which local ATomspaces, and should be brought to the attention of which cognitive processes
- Interfaces to external libraries (e.g. neural net frameworks) that AI agents can use to inform their interactions with Atomspace

Among the numerous questions that arise in this process of creating a detailed design according to this high level architecture are the following *Core Issues*:

1. What should the core Atomese formal language be?
2. Algorithmic approach to Atomese interpretation/compilation
3. Utilization of core Atomese to support various specialized formal languages useful for various AI algorithms
4. Surface form of Atomese language (”syntactic sugar”)
5. RAM-based local Metagraph store – which must be optimized for heavy Atomese usage of certain sorts
6. Distributed and persistent Metagraph store, perhaps with distributed RAM-based middleware as well

7. Atomese libraries corresponding to particular AI algorithms and approaches (e.g. the ones involved in OpenCog already)

8. Mode of integration of Atomese programs with external data/knowledge stores and processing and learning frameworks (e.g. external deep neural net libraries)

3 The Concept of an "Atomese 2" Language

First of all, how and why do we get from talking about AGI to talking about programming languages?

The history of AI has persistently featured fascinating feedback, synergy and tension between AI system design and programming language design. Numerous researchers have come to the conclusion that, to make the radical AI advances they sought, they would require a better and more AI-friendly programming language environment. Thus we got languages like LISP and Prolog and their derivates. Which have taught us a lot about AI and programming, yet without leading so far to the hoped-for AI breakthroughs.

Contemporary neural net based AI hasn't focused on introduction of new full-fledged programming languages, but rather on new libraries such as Tensorflow, Torch, Theano and so forth - which can be considered as DSLs or embedded languages with their own execution graphs, variables, interpreters and so forth. On the other hand, the probabilistic programming paradigm has led to a remarkable profusion of new languages, along with work on the critical underlying issues of efficiently executing probabilistic programs applied to real-world situations.

If one wants to pursue an integrative, multi-paradigm approach to AGI today, then the situation as regards programming languages remains very far from optimal. If one want to integrate, say, a logic programming system with a deep neural net perception system and a program learning system based on higher order functional types – one is quite likely to want to implement the three components in different languages, and glue them together with scripts written in a simple language such as python. Either that or one decides to value consistency and unity over elegance and efficiency, and shoehorns all three into a single language, reconciling oneself to either dramatic inefficiency or unwieldy, awkward code. (The current OpenCog approach, when one wants to combine Atomspace-based operations with operations in external frameworks and systems, often achieves an impressive combination of inefficiency and awkwardness.)

A key and multidimensional point regarding programming language for AGI is the distinction and relationship between languages on four different levels:

1. **Infrastructure Development Language:** Programming language for humans to use to implement the low-level infrastructure of an AGI system

2. **AI Scripting Language:** Programming language for humans to use for creating new AI functions to operate within an AGI system

3. **Engineered Internal AI Language:** Programming language, supplied by humans, for the AI itself to use to represent what it learns, and to guide its learning

4. **Emergent Internal AI Language:** Programming language that is learned by the AI for its own use in representing what it learns and for guidi ng its own learning

The discussion here will be focused mainly on Levels 2 and 3.

However, Level 4 (emergent languages of thought) is never far from my mind, and is obviously heavily conditioned by Level 3, as well as by the specific AI algorithms implemented Level 3 and Level 4. The framework we will propose here for Levels 2 and 3 is sufficiently general and flexible that, via coupling it with paraconsistent reasoning systems and gradual-typing-friendly program learning systems of suitable effectiveness, one should be able to create systems capable of fostering a variety of creative possibilities on Level 4. The "Cogistry" approach to inference-enhanced algorithmic chemistry, and current practical work being done using OpenCog for inferring grammars from the dynamical trajectories of complex networks, are both gestures in the direction of Level 4.

Level 1 (infrastructure language) will not be discussed much here explicitly, so it is probably worth noting that we don't assume any overlap between Levels 1 and 2. The implicit default choice in this context is to use C++ as the infrastructure language for OpenCog Hyperon, just as was done for the original OpenCog, but at time of writing this is not yet a firm decision and the matter is open for analysis and discussion.

3.1 Current "Atomese"

What we currently refer to loosely as an "Atomese program" is a subgraph in Atomspace, each atom in which can be evaluated or executed (interpreted). Basic manipulation of nodes, links, and values (like inserting or deleting them) is on a lower level of Atomspace storage. The Atomese "interpreter" in the current OpenCog system is largely embedded into the Pattern Matcher, but in part distributed across 'execute' and 'evaluate' methods of different Atom, and is ultimately not that different from interpreters of other languages, though it has some special features like BindLink, which are not available in many other languages.

The current Atomese version doesn't have a well-specified formal semantics (computational model), and has been under ongoing experimental development. However, Atomese programs are currently heavily used for manipulating nodes and links and values in OpenCog's Atomspace, and carrying out a few other related operations like sending and receiving Atoms from outside sources and launching processes to manipulate Atoms.

The creation of Atomese programs by humans is currently a bit awkward – Pattern Matcher queries and Atom evaluation and execution invocations are accessed via Atomspace APIs exposed in Scheme (mostly), C++, python or Haskell (where the interface for the latter is quite incomplete). So in a practical programming sense, today's "Atomese" is basically use of Scheme/python/Haskell scripts to manipulate Atomspace APIs, including critically the API for using the Pattern Matcher.

Just to give a flavor to the reader without OpenCog experience (who will probably be a very confused reader of this document generally anyway, but...) – a simple example of Atomese 1 usage is given in Figure 3, drawn from an application built by

Figure 2: Visual example of jaywalking that is recognized by Atomese expression in Figure 3

Cisco Systems [3] in collaboration with SingularityNET Foundation, applying OpenCog to fuse results from multiple vision-processing deep neural nets to make inferential judgments about street scenes.

3.2 Onward to Atomese 2

Our intention is that in creating an updated version of the OpenCog system, an OpenCog Hyperon design, the current loosely defined Atomese "language" will be replaced by a more fully featured "Atomese 2" language, which has a syntactically sugared form making it conveniently usable as an AI scripting language (Level 2 above) and also a non-sugared form appropriate for use as an Engineered Internal AI Language (Level 3 above).

In thinking about how to create such a language, we need to think about issues such as:

- simple representation of the various Atom types in play in the OpenCog design
- effective representation of external entities like knowledge-stores, specialized learning algorithms, simulations etc. as monads [or using some other powerful mode of encapsulation]
- breaking down pattern-matching process into simple atomic operations like "match this pattern at this location" and "move locus of pattern-matching to?"
- compatibility of the above breakdown w/ concurrent and distributed processing
- timed pattern matching should be fairly easily efficiently implementable

[3] https://www.youtube.com/watch?v=s7EtRJatVmg

```
BindLink(
    TypedVariableLink(VariableNode("pedestrian-tracklet"),
                      TypeNode("GroundedObjectNode")),
    AndLink(
        InheritanceLink(VariableNode("pedestrian-tracklet"),
                        ConceptNode("tracklet")),
        InheritanceLink(
            VariableNode("pedestrian-tracklet"),
            ConceptNode(str(RoadUserType.PERSON))),
        ApplyLink(MethodOfLink(
            GroundedObjectNode("traffic-lane"),
            ConceptNode("is_at")),
            ListLink(VariableNode("pedestrian-tracklet")))),
    MemberLink(ConceptNode("jaywalking"),
               VariableNode("pedestrian-tracklet"))))
```

Figure 3: Atomese expression that recognizes simple forms of jaywalking based on output of deep neural visual recognizers. Application of the expression is mediated by OpenCog's URE rule engine, which leverages OpenCog Pattern Matcher internally.

- How can we make it simple for a developer to add low level optimized support for some particular set of predicates / schema that's of interest?

4 Some Considerations Regarding Pattern Matching in Atomese

A peculiarity of the intended AGI use-case for Atomese is that we can assume the vast majority of processing time is spent on two key operations,

1. checking a particular (generally small) sub-metagraph to see if a certain pattern is matched there, *for a wide variety of patterns, to be dynamically generated and not foreseeable in advance*

2. applying a small set of metagraph rewrite rules to a particular (generally small) metagraph

There can be assumed to be roughly comparable balance between these two sorts of operations.

Also, there is a need both for rapid processing of these sorts of queries on a large metagraph in local RAM, and for distributed processing of these sorts of queries on a metagraph that is stored across numerous machines.

This means it is not important that Atomese be especially efficient at, say, sorting lists or computing the FFT. What is important is that it is efficient at doing the above

two operations and piping around, and doing simple manipulations on, the results of these operations. If e.g. list-sorting or mathematical calculations are needed, it is assumed that Atomese will (at least in many cases) get these things done via referencing libraries coded in other languages. Elegant and efficient interfacing with a variety of other languages and toolkits is thus highly important.

Though, it's relevant to note that operations as list sorting will be perfectly possible to code in Atomese 2 – although such implementations will be not as efficient as in ordinary general-purpose languages, because they will be operating on the cognitive/reasoning level. And it will also be feasible, as the system is further developed, to compile Atomese 2 expressions corresponding to deterministic reasoning chains into some more efficient form – so that, for instance, if an AI reasoning or program-learning algorithm learns a new sorting algorithm, it may first represent this sorting algorithm in a way that is practically inefficient due to Atomspace overhead (even if highly algorithmically efficient), but then afterwards compile the algorithm into a highly efficient executable form.

4.1 Decomposing the Pattern Matching and Rule System Execution Process

In OpenCog 1, the two key operations mentioned above are packaged up into the Pattern Matcher, which embodies a particular search algorithm and a variety of programming-language mechanisms along with basic pattern-matching functionality; and the Unified Rule Engine which executes a set of rules using forward or backward chaining, using the Pattern Matcher to manage rule application. This is a powerful approach but also can be overly rigid.

One design idea under discussion regarding Atomese 2 is that the interpretation process should break down an Atomese program into small chunks, which will mostly exemplify the two operations mentioned above (local pattern matching and local rewrite rules), plus operations of traversal within the metagraph. Atomese scripts will then combine these chunks in various ways, dispatching some to remote machines as needed. Improved versions of what the current OpenCog Pattern Matcher and URE do would then be implemented at Atomese scripts combining these elementary chunks. In essence, in this approach Atomese scripts will use functional programming constructs to interweave pattern matching with procedural content execution.

A few other particularities of pattern matching in an integrative AGI context are that:

- Static pattern matching must include matching against Atoms representing variables (i.e. variables must be first-class citizens, treated like any other cognitive content)
- It must also include matching of individual query terms against sub-metagraphs (not just individual nodes/links)

It should also be noted that static Atomspace pattern matching via Breadth-First-Search can be implemented so as to efficiently exploit multi-GPU architectures (using Gunrock [WDP+16] or similar tools).

5 A Two-Layer Design for Atomese 2

This section outlines a potential high level approach to Atomese 2 design based on the above concepts.

5.1 Layer 1: A Generic Atomese Core

To enable the flexible exploration needed to work from our current state of knowledge toward a refined AGI design, the Atomese core must be something quite generic – e.g. it must comprise both

- a way of defining/manipulating Atoms (including specifying Atoms that embody rewriting rules for mapping sub-metagraphs into sub-metagraphs)
- a way of defining/utilizing Atom type systems and Atom indexes associated w/ specific Atom types or type-systems. (Note that the type systems defined should be defined within the same metagraph in which the Atoms reside.)

For each Atom type system that one defines, one should be able to plug in a type-checker / type-inference-system.

The Atomese core may then need to be a rather generic gradual-typing framework, that deals with a system involving some Atoms that have incompletely specified or nonspecified types, and other Atoms that are defined w/in specific type systems.

The appropriate implementation strategy for a highly abstract framework of this nature is not entirely clear, and [Goe20a] presents some possibly relevant suggestions regarding core underlying data structures (combinatory decision dags) for use in this context.

5.1.1 Gradual Typing

For background on gradual typing see: [Sie10] [SG12].

While the matter seems not to have been explored theoretically in great detail, it seems intuitive that gradual typing in programming languages should map via Curry Howard type isomorphisms into paraconsistent logics of some sort.

Achieving efficient execution of gradually typed languages is challenging (though not infeasible) because of obvious issues regarding casting between the dynamically and statically types parts of a program [NL18]. However, given the peculiarities of Atomese, this bottleneck may not matter as the pattern-matching bottleneck may be more severe.

5.2 Layer 2: Initial Type Systems Atop the Core

The next larger layer of the onion would then be a specific type system (or small set of type systems) that we find to be interesting and potentially adequate for the particular AGI-oriented algorithms we're developing in practice. This would be a set of languages / formal systems developed on top of core Atomese. This is where, tentatively, it seems probabilistic linear dependent types will come in.

The obvious advantage of this sort of layered approach is that we can then modify the "probabilistic linear dependent types" or other specific formalizations a little later without having to rebuild the architecture. However, we should expect that in practice nearly all users are going to end up working with the "specific type system" layer of the onion we initially create, rather than the "generic gradual-typing based Atom and Atom-type-system framework" layer.

5.3 Some Specific Type Systems of Apparent AGI Relevance

One hypothesis that seems very much worth exploring is to use *probabilistic linear dependent types with IsoType type inference* as a formalization on top of Atomese core, with power to drive both probabilistic logic and also related applications such as probabilistic program learning.

It seems that this particular flavor of type system may meet the needs of a variety of AI algorithms currently existing in OpenCog, plus others that have been proposed for OpenCog integration: Probabilistic Logic Networks, surprisingness-based pattern mining, probabilistic evolutionary program learning (MOSES), probabilistic programming (including cases with neural nets or probabilistic logic inference on the back end), nonlinear-dynamical attention allocation, content-addressable episodic memory, neural-symbolic perception processing and action control.

The full argument why this particular formalization direction is valuable for meeting these needs of these AI algorithms is involved with many parts and would be too lengthy to full elaborate here. Rather, here only a few of the more critical points will be sketched.

5.3.1 Dependent Types

Dependent types are valuable in an AGI context because they enable elegant manifestation of the morphism between declarative knowledge (logic expressions) and procedural knowledge (programs). Programs expressed with dependent types can be very straightforwardly interpreted as logic expressions. Converting between procedural and declarative knowledge is key to AGI, and having a formalism that makes this convenient is high value. An elegant prototype interpreter for lambda calculus with dependent types is Lambda-Pi, available as open source code on Github [4].

5.3.2 IsoType Systems

Pure IsoType Systems (PITS) are a way to get (a lot of) the power of dependent types without making type-checking undecidable [YO19]. They may also help with making dependent type checking not only decidable but reasonably fast, though this is still an active research topic. Whether their limitations are important from an AGI perspective is not clear.

[4] `https://github.com/lambda-pi-plus/lambda-pi-plus` , `https://github.com/tdietert/lambda-pi`

5.3.3 Linear Types

Engineering General Intelligence [GPG13a] [GPG13b], the foundational book outlining the theory behind OpenCog, has a whole section on "effort management" – counting the computational resource usage of each cognitive operation and using this in planning etc. This is important and ties into Occam's Razor heuristics which are key to AGI theory. though we haven't dealt with explicit effort management much in our practical OpenCog work so far.

Linear logic basically lets you count resource usage in the guts of your logic engine (or equivalently, your program execution process). Of course there are always other ways to do this, but having it built into the logic is a way that fits naturally with reflection and meta-computation. Dependent types have been gotten to work with linear types [KPB15]; and pattern matching with linear types has also been explored [SNS10].

We note that to make probabilistic linear lambda calculus confluent you choose either call-by-value or call-by-reference. Similarly making either of these choices renders type checking decidable in dependent type theory w/ isotypes. One guesses that making probabilistic linear lambda calculus with dependent linear types, if one wants to restrict type equivalency to isotyping, then one will get both confluence and decidability from either choice of call-by-reference or call-by-value.

5.3.4 Probability/Logic Interoperation

Probabilistic methods are probably the biggest innovation in AI over the last couple decades, and it seems clear that including probabilistic representation and manipulation at the basic level is going to be a good idea for any AGI engine.

Recent work [FR19] gives a variant of probabilistic lambda calculus that is confluent (it achieves confluence by limiting the reductions that can take place, in a manner framed via linear logic).

This sort of low-level probability/logic integration lays the groundwork for specific probabilistic-logic math aimed at deductive, inductive, abductive and other forms of inference, such as e.g. OpenCog's Probabilistic Logic Networks (PLN) framework [GIGH08] carries out.

PLN depends heavily on non-confluent reductions in probabilistic logic expressions, however these are necessarily going to be kind of heuristic and history-guided, so it makes sense for them to live in the next layer of the onion – i.e. we have core Atomese, then probabilistic linear dependently types lambda calculus or similar built on that, then PLN built on that. But the building of PLN on top of elaborated lambda calculus can use the same basic Atomese syntax and interpreter as the building of elaborated lambda calculus on core Atomese.

In the context of the above figure, e.g. PLN logic might end up using a type system founded in probabilistic linear dependent types with IsoType type inference. On the other hand, for automated program learning it might be decided that the IsoType approach is too restrictive, and it's better to bite the inefficiency bullet a little harder and go with a more flexible type inheritance mechanism.

In this case, via the gradual typing approach, we could have some Atoms that are

not typed at all, and can thus play a role in either the PLN or program learning focused type systems. On the other hand, if program learning generates a program that then needs to be reasoned about, this will necessitate a mapping from the program-learning type system to the probabilistic-logic type system. There will be some equations that are consistent in one of these logics but not the other (in particular, perhaps some that are consistent using IsoTypes and not using more flexible inheritance mechanisms) – thus rendering the overall framework paraconsistent, rather than strictly consistent.

5.4 How Deeply to Bake In Tensors?

Another important question regarding Atomese 2 / Atomspace 2 design is how to handle the integration between symbolic inference / program-execution and scalable manipulation of large vectors, matrices and more general tensors. The two main alternatives are

- Rely on external libraries for scalable tensor manipulations, and create efficient interfaces between Atomspace based processes and these external libraries (basically, this would be a smoother and more efficient version of what happens in OpenCog now, e.g. in the "Cognitive modules" work by Alexey Potapov et al [PBBS19], but with substantial improvements due to the redesign that more elegantly splits dynamic Atomese-program execution from static pattern matching).
- Embed scalable tensor manipulations deep in the guts of Atomese / Atomspace (so that if third-party libraries are used, they are accessed by the Atomese interpreter/compiler at the internal infrastructure level), so that back-and-forth between symbolic representations of tensor operations and the actual tensor operations occur "behind the scenes" in Atomese interpretation/compilation

Our current inclination is that we will likely need to integrate tensors at the core of Atomspace somehow. Otherwise we suspect there will be ongoing struggles with efficiency and/or awkwardness issues in neural-symbolic operations.

One can represent tensor algebra elegantly and simply using higher order types, e.g. `https://hackage.haskell.org/package/HaskellForMaths-0.4.9/docs/Math-Algebras-TensorAlgebra.html` – and in some ways this representation can benefit from a free-er approach to type manipulation than Haskell allows (see `http://haskellformaths.blogspot.com/2011/07/tensor-algebra-monad.html`). It may be that for Atomspace 2 we want an abstract representation of tensors and related math roughly along these lines, and then the ability to ground tensor objects in some memory-efficient and rapidly-manipulable data structure under the hood. In this approach any back-and-forth between low-level tensor data structure manipulations and abstract tensor-algebra manipulations would be implemented "under the hood" in the guts of the Atomese interpreter for maximum efficiency.

6 Distributed Atomspace

[5]

Another key driver behind our current push to redesign OpenCog / Atomspace is the need for massive scalability. This has multiple aspects, including use of multi-GPUs for metagraph processing, efficient use of SMP across CPUs for AI algorithms on a single machine, but the aspect we will focus on here is distributed processing in which a single conceptual Atomspace is concurrently processed by a large number of processors resident on a large number of networked machines.

Large-scale distributed processing frameworks have been critical to the practical rollout of ML systems in industry, over the last decade+; and if integrative proto-AGI and AGI systems such as we aim to build with OpenCog are going to be rolled out practically for broad-scale practical impact, it will be highly desirable and perhaps even necessary for OpenCog systems to achieve similar scalability properties.

Put simply, we would like to see Atomspace 2 be roughly as fast and as scalable as competing no-SQL knowledge-stores (preferably more so), while retaining its basic flexibility and suitability as an AGI platform.

In this section we explore issues pertaining to making a distributed system for doing Atomspace-API-type stuff against a logical Atomspace that is much too large to fit into the RAM of any one of the machines in the network at hand.

A more in-depth document titled "Distributed Atom Space - Requirements and Ideas" covers many of the topics mentioned here in more details, and also comes along with multiple associated use-case descriptions drawn from experience with OpenCog over the years. What is provided here is more of a high level overview of the issues involved.

6.1 Key Concepts

Discussion of Atomspace in a distributed systems setting can be confusing for basic terminological reasons as well as for more fundamental algorithmic and conceptual reasons. We therefore start by making some basic terminological clarifications regarding Atoms in the various forms they need to exist in a fully distributed OpenCog system.

- Atom - the theoretical abstract definition of the most elementary entity in an AtomSpace, i.e. a node (uniquely identified by its type and name) or link (uniquely identified by its type and outgoing set).
- Realized atom or simply "clone" - atoms sitting in RAM. Thus, given a single, pure "abstract/theoretical" atom, there might be two different realized atoms on two different servers, having the same name/outgoing-set.
- Serialized atom - analogously, we may call atoms living on disk, or flying on a wire, "serialized atoms".

[5] This section draws heavily, including some cut and pasting, from prior larger documents written by Alexey Potapov and Andre' Senna

- Chunk - a subgraph of nodes and links with a self-contained semantic meaning. The actual form and size of a chunk vary from application to application (or even in the same application) but typically it's a link and the whole subgraph defined by it and its constituents. E.g.:

```
(EvaluationLink
    (PredicateNode "predicate1")
    (ListLink
        (ConceptNode "concept1")
        (ConceptNode "concept2")
    )
)
```

  In degenerate cases, a chunk could be solely a node or a single link between two nodes.

- Handle - Locally (in the scope of a single server) unique ID of an atom
- UUID - Universally (in the scope of all servers) unique ID of an atom.
- STI - Short-term importance.
- LTI - Long-term importance.
- DAS - Distributed AtomSpace. A set of atoms stored (RAM and/or disk) across several servers.

6.2 Provisional Assumptions and Constraints

Next we highlight some core practical assumptions we are making regarding distributed Atomspace design and deployment in the next stage.

1. We consider a distributed system composed of multiple components. In a simple case the components may be servers in a network, but we may also want to consider other cases. Each component will contain one or more Atomspaces which are considered to be local to each other.
2. Atoms have an Universally Unique ID (UUID) based in a hash function of its constituents (collisions are allowed but the hash function is such that collisions are rare). Because UUID is computed based in atom's constituents it is unique among all components
3. If useful, it is OK for Atoms to also have a "handle" which is a locally unique id (unique in the scope of a single component). It's possible to use only UUID but in practice using local handles may be better because they are smaller objects and their use causes a significative gain in memory usage.
4. In addition to the information that defines an atom (type, name and outgoing set) atoms may carry additional information (e.g. attention values and truth values which are lists of numbers)

5. Atoms tend not to be that large; a node may be a few dozen to a few hundred bytes.

6. Atom creation and deletion and modification are common events, and occur according to complex patterns that may vary a lot over time, even for a particular OpenCog instance.

7. Some atoms will remain around for a really long time, others will be ephemeral and get removed shortly after they're created.

8. For most of OpenCog's cognitive tasks, the Atoms involved in thinking need to live in RAM. But the saving/retrieving of Atoms to/from disk also has a role to play.

6.3 Tentative Requirements

The above assumptions, together with the nature of the OpenCog AI algorithms currently in use or envisioned for near future use and the practical requirements of near-term OpenCog applications (e.g. humanoid robotics, sandbox-game AI, natural language dialogue, interactive biomedical research,...), lead to a series of fairly clear functional requirements at a medium level of granularity:

1. Two clones of the same atom may be present in different components (if this proves useful; some designs would not require this); but in any case, each component may have only one clone of a given atom. What this means is that within a single component, two processes acting on a certain atom can be confident they are acting on exactly the same thing.

2. There should not be local collision of UUID, i.e., if an atom being inserted in DAS has an UUID collision with another atom in the same server, this collision must be solved immediately.

3. UUID collisions between atoms in different components must be solved the first time one of the atoms is retrieved as answer for a request made in the server where the other atom resides.

4. DAS is supposed to be capable of storing an "as large as necessary" number of atoms. So it can not rely on the number of servers to provide this feature since this number may be restricted by deployment constraints. So DAS must use disk or DB persistence as additional/optional storage resources.

5. Decision of which atoms should be serialized and which should be kept in RAM must be guided by STI and LTI, so that the more important Atoms are most likely to be kept in RAM.

6. User may explicitly request the serialization of an atom or a set of atoms to save room in RAM for more important information.

7. Optionally, user may explicitly restrict the scope of a newly created atom to the local component. This should (completely) prevent this atom from being returned as answer to requests made in other components.

8. User may request DAS to move sets of atoms to a specific component, allowing local access for AI algorithms in that component.

9. User may request DAS to freeze sets of atoms in a given component, preventing than from being moved by the underlying balance strategy of DAS.

10. The balancing strategy of DAS should seek to minimize the overall processing time required for the whole distributed network to carry out its AI operations. This may mean moving atoms around among components, or creating clones of atoms in different components, in order to minimize the amount of time spent by AI algorithms running in one component needing to access atoms living in other components.

11. Two (or more) servers may create or update a realized atom (e.g. by changing its truth value) at the same time. Potentially this can cause inconsistency in atom's state. DAS is supposed to solve this inconsistency accordingly.

    - Inconsistencies should be solved by heuristics appropriate to the semantics of the atoms involved, e.g., by PLN's Rule of Choice (which invokes the PLN revision rule much of the time).
    - There is a trade-off between the overall consistency of the atoms in the whole DAS and total processor time spent to keep this consistency. This trade-off is supposed to be adjustable by the user. One should be able to adjust the system to allow a fairly high degree of inconsistency, more so than in typical distributed data-store applications.
    - It should be possible to tune system parameters to enforce a high level of consistency. (non-functional)
    - It is not important to have high efficiency under conditions of high consistency. We anticipate performance to be more important than strict consistency.
    - It should be very unlikely for any inconsistency in the system to remain forever i.e., the system should make an effort to resolve inconsistencies, although this effort will be balanced against the need for efficiency.
    - Resolution of inconsistencies should be guided by STI and LTI, so that the more important Atoms have their inconsistencies resolved faster.

12. There are use cases where DAS + AtomSpace will be used like an hierarchical cache structure. User may request a specific Atom (or set of atoms) to DAS and it should search for it (them) locally before trying to fetch it (them) from other component(s), disk or DB.

    - Return the handle of an atom given an UUID. Returned handle is supposed to be consistent in future requests i.e. if the user make a subsequent request passing the same UUID, the same handle is supposed to be returned.

- Return the handle of an atom given its definition (type + name for nodes or type + outgoing set for links).
- Again, returned handle is supposed to be consistent in future requests. Completely erase an atom, removing it from whatever storage entity used by DAS (yes, this is just a simple DELETE. The emphasis is to enforce that the erased atom is not supposed to be kept in disk, DB etc)
- Return the handle of a node, given its type and name.
- Return the handle of links with a specified (UUID, handle or type/name/outgoing set) atom as target or source.

13. There are use cases where DAS will be used like a Database. User may request for atom(s) that satisfy a given property so DAS should search locally and remotely for the proper query answer.

    - Optionally, in all the requests below, the user may be interested only in realized atoms, i.e. atoms that have not being serialized to disk or DB.
    - Optionally, in all the requests below, the user may be interested only in atoms residing in local component.
    - Return handles of all atoms of a given type.
    - Count the number of atoms of a given type.
    - Return all variable assignments that satisfy a given pattern, as defined in the PatternMatcher API.
    - Return handles of the M atoms with highest/lowest LTI or STI. Optionally constrained to a given atom type.
    - Return handles of all atoms with time-stamp within a certain time-interval.
    - Return handles of all atoms with spatial location within a certain region, during a certain time-interval.

14. DAS must be able to consider chunks of atoms, rather than just individual atoms, as elementary units to be distributed amongst components. The actual form/size of a chunk is application-dependent, and should thus be user-configurable.

A number of associated non-functional requirements are also fairly clear at this stage:

1. We want 5 to 10 million atoms on each server to allow AI algorithms to run in a fast and sensible way.
2. Typically we want clusters as large as dozens of servers.
3. We also want to be able to have *networks of clusters* that occupy massive data centers. (But the relationship between machines in two different clusters, may be totally different than the relationship between machines in the same cluster.)
4. The system should be able to run on a cluster of roughly equally powerful machines.

5. In addition to the atoms storage itself, each server in the cluster may run one or more AI algorithms potentially using local and remote atoms. Such algorithms tend to create/use information locally (in the hosting server) but not necessarily.

6. User's requests to DAS that can be answered without inter-server communication should be answered in the same (or nearly the same) time as requests to a regular AtomSpace. In other words, DAS is supposed to add very small overhead for requests that can be answered exclusively with local information.

7. User's requests to DAS that demands inter-server communication should be answered at a rate of at least 5 user requests/second.

6.4 Handling Inconsistencies

One of the subtler issues in DAS design is the handling of inconsistencies that will arise in the operation of massively distributed mind. This is an area where cognitive and logical matters intersect closely with distributed implementation issues. OpenCog's PLN logic is paraconsistent by design, not requiring global consistency; and gradual typing, which is our recommendation for the dependent type framework for Atomese 2, is the programming-language cognate of paraconsistency in the logic domain. In typical distributed system designs, a lot of work is expended on guaranteeing consistency (so that e.g. the copy of my Facebook profile on one of Facebook's servers is identical to the copy on another of Facebook's servers), however in an AGI context, there are some cases where maintenance of precise consistency is important, and others where it is not and it can be allowed to slip in order to optimize other parameters of interest (e.g. speed of drawing interesting approximate conclusions).

With this in mind, it may be sensible to make a DAS that can

- enforce strong consistency in certain "zones" of the DAS
- allowing a roughly controllable amount of inconsistency in other zones

With this in mind we will now sketch a design that could serve to enforce an approximate consistency on DAS, without seeking a perfect continual consistency. Connected closely with this, we also describe the atom migration and cloning process, which deals with movement of atoms between servers and creation of atom clones.

The key issue to be dealt with in this context is: What happens if two different servers get Atoms which have the same (name, type) pair, or the same (type, target list) tuple?. The approach suggested here is based on the idea that , in the "partially consistent" zones of the DAS, we should be OK with this situation existing for a little while. In these DAS zones, we're not aiming to make a wholly mathematically precise reasoning system, we're aiming to make the more creatively and informally reasoning portions of a generally intelligent mind.

In partially consistent DAS zones, inconsistencies of the type mentioned just above could be handled as follows.

- Each server should put each newly created realized atom inserted into it, into a newAtoms structure. This structure should have indexes ordering it by STI, LTI,

creation time, and Handle. When a realized atom is removed from the server, a check is made whether it's in the newAtoms structure or not. If so, it's removed.

- Each server run an instance of the AtomReconciliation agent, which is an OpenCog's MindAgent.

In this approach, each time the AtomReconciliation agent is called, it processes a certain number of realized atoms in the newAtoms structure. It chooses the realized atoms to process based on a combination of their STI, LTI and recency of creation. Highest priority is given to realized atoms with higher STI and LTI that have been around longer. Lowest priority is given to realized atoms with low STI and LTI that are recently created. Realized atoms with high STI and recency but low LTI, also should not be called in a big hurry (many perceptual atoms will be like this, and they don't strongly need to get reconciled).

This algorithmic direction has been sketched in greater detail, here the goal has just been to provide an outline, partly just to indicate the kind of non-traditional thinking that may be needed to make a usably efficient and massively scalable DAS a reality.

6.5 Potential Implementation Directions

Implementing a massively distributed metagraph store from scratch would be a huge job, and there is a strong motivation to draw on existing scalable graph DBs, tuple stores, and so forth.

However, when one digs into the details, it appears that graph DBs will not be appropriate for the DAS, and it will be necessarily to dig one level deeper and leverage existing key-value stores instead. A detailed analysis of the situation is given in the document "Distributed Atomspace 2.0 and graph DBs: differences", here we will provide a brief summary.

Roughly, it seems that graph DBs are too general in some senses (because they are made to apply to arbitrary graphs) and not general enough in other senses (meaning that that they don't really cover the needs of Atomese). If in OpenCog work we need to deal with data for which graph DBs are well-suited, we can just use such DBs as concrete containers within Atomese; but it seems this won't be the generic or even the most common case.

This brings us back to the in depth discussion of pattern matching from earlier sections. We believe it will be possible to have a restricted pattern matching at the core of Atomese and to make only this restricted pattern matching distributed – leaving most more sophisticated Atomese code running on individual machines, making calls to a restriction pattern matching library with a distributed implementation. However, even the restricted pattern matching that we would need to make distributed in this approach, will still requires some features which are not supported by the existing graph DBs. For instance

- An advanced generic graph traversal API (more typical for graph dbs) is not of highest priority
- Deduplication of subgraphs (which complicates solutions of some tasks over ordinary graphs) is still a default choice for many Atomspace tasks

- Representation and retrieval mechanisms should be biased towards acyclic hypergraphs (and related more general sorts of metagraphs) instead of ordinary graphs, in particular, grounding of variables to a whole subgraphs (expressions) at once is to be supported
- Variables in KB should be supported in contrast to graph DBs where these are not part of the infrastructures
- Pattern matching should ground variables in KB entries and queries simultaneously
- Our query language (Atomese 2) must be introspective (that is, the KB should be able to keep expressions in the query language itself, and the query engine should be able to interpret them)

We could in principle emulate all these features using ordinary graph DBs, but it would be inefficient and cumbersome (and in the end not clearly preferable to using a traditional SQL based relational database, which could also emulate all these things with sufficient contortions).

It seems an appropriate solution may be to build our own (meta)graph query engine over a lower-level framework like key-value storage, or to modify something like dgraph for our needs. Badger and RocksDB are examples of existing frameworks that might be appropriate as underlying key-value stores.

Regarding overall architecture, one option might be to connect a local cache Atomspace (perhaps similar in many respects to the current Atomspace) with an appropriately configured distributed key-value store using a middleware solution like Apache Ignite. Ignite has been evaluated for this purpose in some detail and would appear to have the flexibilty to support the various querying and pattern matching requirements mentioned above, as well as the flexible inconsistency management strategies suggested above. But there may be other alternatives as well.

References

[FR19] Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. *CoRR*, abs/1901.02853, 2019.

[GIGH08] B. Goertzel, M. Ikle, I. Goertzel, and A. Heljakka. *Probabilistic Logic Networks*. Springer, 2008.

[Goe20a] Ben Goertzel. Combinatorial decision dags: A natural computational model for general intelligence. In *AGI Conference Proceedings*, 2020.

[Goe20b] Ben Goertzel. What kind of programming language best suits integrative agi? In *AGI Conference Proceedings*, 2020.

[GPG13a] Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 1: A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*. Springer: Atlantis Thinking Machines, 2013.

[GPG13b] Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 2: The CogPrime Architecture for Integrative, Embodied AGI*. Springer: Atlantis Thinking Machines, 2013.

[KPB15] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ?15, page 17?30, New York, NY, USA, 2015. Association for Computing Machinery.

[NL18] Max S. New and Daniel R. Licata. Call-by-Name Gradual Type Theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[PBBS19] Alexey Potapov, Anatoly Belikov, Vitaly Bogdanov, and Alexander Scherbatiy. Cognitive module networks for grounded reasoning. In *AGI*, 2019.

[SG12] Jeremy Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. 09 2012.

[Sie10] Jeremy Siek. What is gradual typing? 2010. `https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/`.

[SNS10] Anders Schack-Nielsen and Carsten SchÃŒrmann. Pattern unification for the lambda calculus with linear and affine types. volume 34, pages 101–116, 09 2010.

[WDP+16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[YO19] YANPENG YANG and BRUNO C. D. S. OLIVEIRA. Pure iso-type systems. *Journal of Functional Programming*, 29:e14, 2019.