OpenCog Prime: A Design for A Thinking Machine

(Draft)

Ben Goertzel

Table of Contents

Preface

These words are being written by Ben Goertzel in late July 2008, as he launches the "OpenCogPrime wikibook" -- a Wiki-fied version of a what originated as a full-length "paper book" manuscript (it was around 250 pages as a Word document) describing a design for a powerful artificial general intelligence, aka a "thinking machine."

The AGI design described in this wikibook, called OpenCogPrime (OCP), is a variant of the Novamente Cognition Engine design that has been refined and developed within Novamente LLC during the period 2001-2008 (and is still the subject of ongoing work).

The reasons for making a wikibook rather than a paper book are probably pretty obvious. The key aspects of the OCP design are not expected to change anytime soon, but the details are ever-evolving ... and this seems likely to continue to be the case for a long time (including after the system implemented according to the design starts evolving itself!!). Because of this, a dynamic wiki, rather than a static paper book, seems the most appropriate medium.

The OpenCogPrime design is proposed here together with the hypothesis that, if it is fully fleshed-out, implemented, tested, tuned and taught, it will lead to a software system with intelligence at the human level and ultimately beyond.

Make no mistake: there is still some research and experimentation to be done in fleshing out the details of OCP AGI design. Some aspects have been implemented in software, others have been fleshed out in detail mathematically and software-design-wise, and others have not yet been explored as thoroughly yet. There is still work to be done. But all aspects of the design have been thought through very carefully over a period of years, and most importantly, the holistic nature of the design has been carefully worked out so that the various components may appropriately interact to give rise to the necessary emergent structures and dynamics of intelligence.

Sister Books

Ideally this wikibook should be read in conjunction with two paper books that were written concurrently,

- The Hidden Pattern (BrownWalker, 2006) ... containing the philosophy of mind underlying the OpenCogPrime design
- Probabilistic Logic Networks (Springer, 2008), giving the mathematical and conceptual foundations underlying the PLN component of OpenCogPrime

Ancient History

The paper book manuscript that formed the raw material for this wiki, itself had a fairly long prehistory, which is worthy of a brief note for those few readers who may have read these previous versions:

In the beginning (2001-2003) there was a big, monster manuscript known informally as "The Novamente Book."

Then, in 2004, this was split into three books:

- The Hidden Pattern (BrownWalker, 2006)
- Probabilistic Logic Networks (Springer, 2008)
- A book describing the overall "thinking machine" design underlying the Novamente Cognition Engine

The latter book was never finalized or published, and eventually evolved into a wikibook on the Novamente LLC internal wiki site, portions of which were eventually morphed into the OpenCogPrime wikibook you are now reading.

A Note on Authorship

The initial version of this wikibook was mostly written by Ben Goertzel, with some help from Cassio Pennachin, Moshe Looks and others. It is hoped that the Wiki version will become more of a group effort, as it is edited over time in accordance with the progress of the project.

The Reader Beware

Though this manuscript has undergone numerous revisions, it is still a draft rather than a finished book. Errors may exist, and several sections are known to be somewhat awkward in exposition due to their origins as cut-and-pastes from multiple prior documents. *Smoothening out* of these things will be undertaken before the book is judged complete.

Also, the formal notation used for various examples in the book is known to be minorly inconsistent throughout the book. This will be regularized in a subsequent revision.

See EditsNeeded for more details on known issues with the wikibook, mainly formatting-related stuff:

Notes on Chapter Ordering and Revisions

The pages in this wikibook can be read in various orders and will probably still make some sense. However, they were originally organized into chapters and arranged in a certain particular order, and are presented that way here.

It will be appreciated if folks who update this wiki will respect this "chapter" ordering. It's fine to introduce new pages into chapters, and to introduce new chapters (even if this requires renumbering existing chapters). But please don't list a page under a chapter heading unless it really belongs there.

Pages should not refer to chapters by number, since chapter numbers may change as new chapters are introduced. The chapter numbers exist only to indicate the ordering of the chapters on this page.

What Kinds of Revisions are Appropriate?

The goal of this wikibook (the OpenCogPrime namespace on this wiki) is to describe ONE PARTICULAR AGI DESIGN, which can be built on top of the OpenCog Framework

Arguments against OCP as an AGI approach belong on the CriticismsOfOpenCogPrime page.

If you want you to describe your own AGI designs or algorithms based on the OpenCog Framework, please start at your userpage and expand from there; please do not use the OCP pages for this purpose.

Improvements and additions to the OpenCogPrime design are appreciated and desired. However, please do not delete existing ideas from the wikibook in order to insert your new ones. Rather, insert your new ideas as alternatives, and label them as such. If you think it's appropriate to delete ideas and design aspects from the book, please consult Ben Goertzel directly; but this will hopefully be done sparingly and only in cases where certain ideas in the book have really been shown to be suboptimal.

Notes on Wikilinks & Namespaces

Three namespaces are used on this wiki, which may seem confusing to the unititated, so here's an example that demonstrates the separation of namespaces, and using pretty wikilinks (mousing over the link displays the namespace):

MindOntology Cognitive Architecture

- a general definition of the concept of a cognitive architecture

Main Cognitive Architecture

- an explanation of how cognitive architectures are implemented using the OpenCog Framework

OpenCogPrime Cognitive Architecture

- an explanation of the specific cognitive architcture in the OCP design

Chapter One: Introduction

This section of the book might be better titled: "Instead of an Introduction, by a man too busy to write one." This Introduction section mostly gathers some relevant information from elsewhere: some links to papers a closely-related AI system (the Novamente Cognition Engine), and some pages from the Mind Ontology. (Much of the Mind Ontology could serve as prefatory material for this wiki, but a couple pages in particular seem strikingly relevant.)

OpenCogPrime, as such, is a new AGI design, but it is a modification of the Novamente Cognition Engine (NCE) design, which was originated in 2001 and has been steadily refined by a group of people since then.

Furthermore the NCE design was itself a (major) modification of the Webmind AI Engine design developed (and partially implemented) at Webmind Inc. during 1998-2000.

Because of this history, the best way to get a general, high-level overview of OpenCogPrime as an AGI design is probably to read some of the previously published conference papers on the NCE. Look for papers authored or co-authored by Ben Goertzel on the following pages:

- http://www.novamente.net/papers
- http://www.agiri.org/workshop
- http://www.agi-08.org/conference

There are differences between the NCE and OCP, but most of these are either

- at a level of detail that's not relevant to these conference papers (e.g. the OCP scheduler is much simpler than the NCE scheduler), or else
- regarding aspects of the NCE that are considered proprietary to Novamente LLC and not disclosed in these conference papers

Probably a nice overview of the OpenCogPrime architecture will be written up in a journal article at some point, but that doesn't seem to be anyone's priority at the moment, and will likely be done after there are some interesting OCP applications and demos to show off and write about.

Finally, the reader is reminded that this wikibook is intended to be read together with

- The Hidden Pattern (Ben Goertzel, Brown Walker Press, 2006), which outlines the underlying philosophy of mind
- Probabilistic Logic Networks (Ben Goertzel, Matthew Ikle', Izabela Goertzel and Ari Heljakka, Springer, 2008), which explains the math and concepts underlying the probabilistic logic component

Patternism (The Patternist Philosophy of Mind)

The patternist philosophy of mind is a general approach to thinking about intelligent systems which I (Ben Goertzel) have developed during the last two decades (in a series of publications beginning in 1991, most recently The Hidden Pattern). It is based on the very simple premise that *mind is made of pattern.*

Now, pattern as the basis of mind is not in itself is not a very novel idea — this concept is present, for instance, in the 19'th-century philosophy of Charles Peirce (1935), in the writings of contemporary philosopher Daniel Dennett (1991), in Benjamin Whorf's (1964) linguistic philosophy and Gregory Bateson's (1979) systems theory of mind and nature. Bateson spoke of the Metapattern: *that it is pattern which connects.* In a series of prior writings (Goertzel, 1993, 1993a, 1994, 1997, 2001) and most recently in the philosophical treatise The Hidden Pattern (2006), I have sought to pursue this theme more thoroughly than has been done before, and to articulate in detail how various aspects of human mind and mind in general can be well-understood by explicitly adopting a patternist perspective. This work includes attempts to formally ground the notion of pattern in mathematics such as algorithmic information theory (Chaitin, 1986; Solomonoff, 1964, 1978) and probability theory, beginning from the conceptual notion that *a pattern is a representation as something simpler* and then utilizing appropriate mathematical concepts of representation and simplicity. In prior writings I have used the term

psynet model of mind to refer to the application of patternist philosophy to cognitive theory, but I have now deprecated that term as it seemed to introduce more confusion than clarification.

In the patternist perspective, the mind of an intelligent system is conceived as the set of patterns in that system, and the set of patterns emergent between that system and other systems with which it interacts. The latter clause means that the patternist perspective is inclusive of notions of *distributed intelligence* (Hutchins, 1996). Intelligence is conceived, similarly to in Marcus Hutter's (2005) recent work, as the ability to achieve complex goals in complex environments; where complexity itself may be defined as the possession of a rich variety of patterns. A mind is thus a collection of patterns that is associated with a persistent dynamical process that achieves highly-patterned goals in highly-patterned environments.

An additional hypothesis made within the patternist philosophy of mind is that reflection is critical to intelligence. This lets us conceive an intelligent system as a dynamical system that recognizes patterns in its environment and itself, as part of its quest to achieve complex goals.

While this approach is quite general, it is not vacuous; it gives a particular structure to the tasks of analyzing and synthesizing intelligent systems. About any would-be intelligent system, we are led to ask questions such as:

- How are patterns represented in the system? That is, how does the underlying infrastructure of the system give rise to the displaying of a particular pattern in the system's behavior?
- What kinds of patterns are most compactly represented within the system?
- What kinds of patterns are most simply learned?
- What learning processes are utilized for recognizing patterns?
- What mechanisms are used to give the system the ability to introspect (so that it can recognize patterns in itself?

Now, these same sorts of questions could be asked if one substituted the word *pattern* with other words like *knowledge* or *information.* However, I have found that asking these questions in the context of pattern leads to more productive answers, because the concept of pattern ties in very nicely with the details of various existing formalisms and algorithms for knowledge representation and learning.

Pursuing the patternist philosophy in detail leads to a variety of particular hypotheses and conclusions. Following from the view of intelligence in terms of achieving complex goals in complex environments, comes a view in which the dynamics of a cognitive system are understood to be governed by two main forces: self-organization and goal-oriented behavior. And more specifically, based on these concepts several primary dynamical principles may be posited, including:

- Association. Patterns, when given attention, spread some of this attention to other patterns that they have previously been associated with in some way. Furthermore, there is Peirce's *law of mind* (Peirce, 1935), which could be paraphrased in modern terms as stating that the mind is an associative memory network, whose dynamics dictate that every idea in the memory is an active agent, continually acting on those ideas with which the memory associates it.
- Differential attention allocation. Patterns that have been valuable for goal-achievement are given more attention, and are encouraged to participate in giving rise to new patterns.
- Pattern creation. Patterns that have been valuable for goal-achievement are mutated and combined with each other to yield new patterns.
- Credit Assignment. Habitual patterns in the system that are found valuable for goal-achievement are explicitly reinforced and made more habitual.

Next, for a variety of reasons it becomes appealing to hypothesize that the network of patterns in an intelligent system must give rise to the following large-scale emergent structures

- Hierarchical network. Patterns are habitually in relations of control over other patterns that represent more specialized aspects of themselves.
- Heterarchical network. The system retains a memory of which patterns have previously been associated with each other in any way.
- Dual network. Hierarchical and heterarchical structures are combined, with the dynamics of the two structures working together harmoniously.
- Self structure. A portion of the network of patterns forms into an approximate image of the overall network of patterns.

Knowledge Representation

In a general sense, *knowledge representation* in AI refers to the patterns in an AI system (or in some cases, patterns emergent between the AI system and its environment) that correlate with patterns the system has experienced, and/or patterns the system is likely to enact or experience in future.

In this very general sense, the easiest way to formally define KR is via reference to an ensemble of AI's (so that, across this ensemble, one can define correlations between experience/enacted patterns and internal patterns).

In many AI designs however, KR is much more explicit than the above description suggests. In traditional, symbolic AI systems, there are explicit, logical, human-comprehensible representations within the system's knowledge base, that reflect the system's prior or future experiences in a transparent way. In AI systems built according to other methodologies, however, there need not be any explicit KR in this sense.

A key point for AGI is that, in an AGI system that is going to remotely approach human-level intelligence, knowledge representation is necessarily going to be mixed up with *meta knowledge representation*: the ability to create new contextually-appropriate representations. For instance, if an AGI is taught to play chess, rather than being programmed to do it, it's going to have to somehow conceive its own representation scheme(s) for internally representing chess pieces and their relationships. If an AGI is taught language, rather than having its linguistic knowledge fully preprogrammed, then it's going to have to somehow conceive its own representations for the linguistic constructs it has learned. And so forth. Specific domains generally demand specialized KR, in order to enable reasonably efficient domain-specific intelligent action selection; and an AGI by its nature cannot have preprogrammed specialized KR for every domain it confronts.

In particular, the KR scheme of an AGI that deals with the same sorts of problems that the human mind does, must be able to handle meta-KR in domains ranging across perceptual, motoric, procedural, declarative, episodic, abstract, inguistic, introspective, etc.

Four Kinds of Knowledge

One powerful approach to knowledge representation (and the one taken in OpenCog Prime) is to decompose knowledge into 4 major subtypes:

1. Sensory = data coming into the system

2. Procedural = actions carried out by the system (internally or externally)
3. Episodic = memories of the system's experience
4. Declarative = knowledge abstracted from the system's experience, actions and sensations

While the OpenCog framework does not enforce this distinction, it does provide useful tools for managing these types of knowledge in a specific way. For instance

1. the OpenCogPrime:AtomTable is a flexible method of storing declarative knowledge, as well as some sorts of sensory, procedural and episodic knowledge
2. the OpenCog Procedure Repository is a flexible method of storing Procedures that are represented as simple LISP-like program trees (note that the index of a procedure in the repository is the Atom Handle of the OpenCogPrime:Node representing the OpenCogPrime:Procedure in the OpenCogPrime:AtomTable.

The OpenCogPrime:AtomTable as a software structure may be used in many different ways. As used in OpenCog Prime, it bridges the gap between subsymbolic (neural net) and symbolic (logic / semantic net) representations, achieving the advantages of both, and synergies resulting from their combination. It is a container consisting of nodes and links (a weighted, labeled hypergraph), which are labeled with both

1. Probabilistic weights, like an uncertain semantic network (these are OpenCogPrime:TruthValue objects
2. Hebbian weights, like an attractor neural network (these are OpenCogPrime:AttentionValue objects

It is not quite a neural net, not quite a semantic net: the best of both worlds!

In OpenCog Prime, episodic knowledge is stored largely via special extensions to the AtomTable such as the OpenCogPrime:SpaceServer and OpenCogPrime:TimeServer, together with the Mind's Eye internal simulation environment. The Space and Time servers allow Atoms to be indexed by absolute or relative space and time, which allows the AtomTable to be used to store sets of events constituting the outlines of episodes. These episode-outlines may then be internally replayed in the Mind's Eye, thus constituting episodic knowledge.

Sensory knowledge in OpenCog Prime is to be represented as a combination of Atoms and more specialized representations. For instance, if an OCP based system is perceiving polygonal objects in a simulation world, it may be supplied with a PolygonTable which stores a collection of object-descriptions, in which each object is represented as a collection of coordinates (the corners of the polygons that constitute the object). Each perceived or remembered object may then be represented as an Atom, and the Atom Handle of this Atom is used as an index into the PolygonTable, which contains information regarding the polygons making up the object.

This decomposition of knowledge into four primary subtypes appears relevant to the human brain, as evidenced by cognitive science and neuroscience research. It is also natural in terms of existing computer science technologies and mathematical formalisms.

However, one may also isolate additional knowledge subtypes and build in specialized KR mechanisms for them. For instance, there is some literature arguing that the human brain contains special mechanisms for handling social knowledge. In OpenCog Prime, in the current design, this is handled via specialized mechanisms embedded within the AtomTable. For instance, when the system encounters another agent, it creates an OpenCogPrime:AgentNode corresponding to that agent, and then there is a Theory of Mind Agent OpenCogPrime:MindAgent that specificaly updates these AgentNodes. In this case, there is a dynamic distinction between AgentNodes and ordinary declarative knowledge; there is not a dramatic representational

difference on the software level, but there is a dramatic representational difference on the deeper level in which knowledge representation is considered as internal patterns correlating with experienced patterns.

Probability as Glue

While each of the four types of knowledge mentioned above may be represented and manipulated separately, it's also important to be able to convert knowledge between the various types.

For this purpose, having a cross-type mathematical formalism and conceptual framework is highly valuable from an AGI point of view. One powerful approach is to use probability theory as the "lingua franca" enabling integration of the different knowledge types. This is the approach taken in OpenCog.

This approach is fairly well aligned with the mainstream of academic AI research: probabilistic AI of various sorts has been increasingly popular and successful in recent years.

Self-modifying, Evolving Probabilistic Hypergraphs

Patternist philosophy is extremely general, which is both a strength and a weakness. In order to more effectively apply it to the AGI problem, I have created an intermediate formalism called Self-Modifying, Evolving Probabilistic Hypergraphs (SMEPH). SMEPH is a more specific formalism for describing intelligent systems, which is consistent with patternist philosophy but provides more guidance regarding the analysis and construction of particular intelligent systems.

The basic ideas underlying SMEPH are threefold, as the acronym would suggest:

- To use a specific mathematical structure called a *generalized hypergraph* to model intelligent systems.
- To study the way hypergraphs change over time (i.e. the way they *evolve* — the word *evolution* is used here in a general sense, rather than specifically in the sense of evolution by natural selection, although that is an aspect of SMEPH as well when one delves into the details.)
- To use probability theory to study the relationships between the parts of the hypergraph.

A hypergraph is an abstract mathematical structure (Bollobas, 1998), which consists of objects called Vertices and objects called Edges, which connect the Vertices. In computer science, a *graph* traditionally means a bunch of dots connected with lines (i.e. *vertices* connected by *edges*, or *nodes* connected by *links*). A hypergraph, on the other hand, can have Edges that connect more than two Vertices; and SMEPH's hypergraphs extend ordinary hypergraphs to contain additional features such as Edges that point to Edges instead of Vertices; or Vertices that, when you zoom in on them, contain embedded hypergraphs. Properly, SMEPH's hypergraphs should always be referred to as *generalized hypergraphs,* but this is cumbersome, so we will persist in calling them *hypergraphs* instead. In a hypergraph of this sort, edges and vertices are not as distinct as they are within an ordinary mathematical graph (for instance, they can both have edges connecting them), and so it is useful to

have a generic term encompassing both Edges and Vertices; for this purpose, in SMEPH and OCP, we use the term *Atom.*

A *weighted, labeled hypergraph* is a hypergraph whose Edges and Vertices come along with labels, and with one or more numbers that are generically called *weights.* The label associated with an Edge or Vertex may sometimes be interpreted as telling you what *type* of entity it is. On the other hand, an example of a weight that may be attached to an Edge or Vertex is a number representing a probability, or a number representing how important the Vertex or Edge is to the system.

Hypergraphs may come along with various sorts of dynamics. Minimally, one may think about:

- Dynamics that modify the properties of Vertices or Edges in a hypergraph (such as the weights attached to them.)
- Dynamics that add new Vertices or Edges to a hypergraph, or remove existing ones.

The SMEPH approach to designing and implementing intelligent systems is centered on a particular collection of Vertex and Edge types. The key Vertex types are

- ConceptVertex, representing a set — for instance, an idea or a set of percepts
- SchemaVertex, representing a procedure for doing something (perhaps something in the physical world, or perhaps an abstract mental action).

The key Edge types are

- ExtensionalInheritanceEdge (ExtInhEdge for short: an edge which, linking one Vertex or Edge to another, indicates that the former is a special case of the latter)
- ExtensionalSimilarityEdge (ExtSim: which indicates that one Vertex or Edge is similar to another)
- ExecutionEdge (a ternary edge, which joins {S,B,C} when S is a SchemaVertex and the result from applying S to B is C).

So, in a SMEPH system, one is often looking at hypergraphs whose Vertices represent ideas or procedures, and whose Edges represent relationships of specialization, similarity or transformation among ideas and/or procedures.

ExtInh and ExtSim Edges come with probabilistic weights indicating the extent of the relationship they denote (e.g. the ExtSimEdge joining the *cat* ConceptVertex to the *dog* ConceptVertex gets a higher probability weight than the one joining the *cat* ConceptVertex to the *washing-machine* ConceptVertex). The mathematics of transformations involving these probabilistic weights becomes quite involved — particularly when one introduces SchemaVertices corresponding to abstract mathematical operations, a step that enables SMEPH hypergraphs to have the complete mathematical power of standard logical formalisms like predicate calculus, but with the added advantage of a natural representation of uncertainty in terms of probabilities, as well as a natural representation of networks and webs of complex knowledge.

Derived Hypergraphs

SMEPH hypergraphs may be used to model and describe intelligent systems (such as human mind/brains, for example). One can (in principle) draw a SMEPH hypergraph corresponding to any individual intelligent system,

with Vertices and Edges for the concepts and processes in that system's mind. This leads to what is called the *derived hypergraph* of that system.

SMEPH Vertices

A ConceptVertex in the derived hypergraph of a system corresponds to a structural pattern that persists over time in that system; whereas a SchemaVertex corresponds to a multi-time-point dynamical pattern that recurs in that system's dynamics. Drawing the derived hypergraph of an intelligent system is one way of depicting the mind of that system — this follows from the definition of a mind as the set of patterns in an intelligent system, and the fact (which follows from mathematical pattern theory) that the patterns in the system can be read off from the derived hypergraph.

To phrase it a little differently, we may say that a ConceptVertex, in SMEPH, refers to the habitual pattern of activity observed in a system when some condition is met (this condition corresponding to the presence of a certain pattern). The condition may refer to something in the world external to the system, or to something internal. For instance, the condition may be *observing a cat.* In this case, the corresponding Concept vertex in the mind of Ben Goertzel is the pattern of activity observed in Ben Goertzel's brain when his eyes are open and he's looking in the direction of a cat. The notion of *pattern of activity* can be made rigorous using mathematical pattern theory, as is described in The Hidden Pattern.

Note that logical predicates, on the SMEPH level, appear as particular kinds of Concepts, where the condition involves a predicate and an argument. For instance, suppose one wants to know what happens inside Ben's mind when he eats cheese. Then there is a Concept corresponding to the condition of cheese-eating activity. But there may also be a Concept corresponding to eating activity in general. If the Concept denoting the activity of eating X is generally easily computable from the Concepts for X and eating individually, then the eating Concept is effectively acting as a predicate.

A SMEPH SchemaVertex, on the other hand, is like a Concept that's defined in a time-dependent way. One type of Schema refers to a habitual dynamical pattern of activity occurring before and/or during some condition is met. For instance, the condition might be saying the word *Hello.* In that case the corresponding SchemaVertex in the mind of Ben Goertzel is the pattern of activity that generally occurs before he says *Hello.*

Another type of Schema refers to a habitual dynamical pattern of activity occurring after some condition X is met. For instance, in the case of the Schema for adding two numbers, the precondition X consists of the two numbers and the concept of addition. The Schema is then *what happens when the mind thinks of adding and thinks of two numbers.*

Finally, there are Schema that refer to habitual dynamical activity patterns occurring after some condition X is met and before some condition Y is met. In this case the Schema is viewed as transforming X into Y. For instance, if X is the condition of meeting someone who is not a friend, and Y is the condition of being friends with that person, then the habitually intervening activities constitute the Schema for making friends.

SMEPH Edges

SMEPH edge types fall into two categories: functional and logical. Functional edges connect Schema vertices to their input and outputs; logical edges refer mainly to conditional probabilities, and in general are to be interpreted according to the semantics of Probabilistic Logic Networks.

Let us begin with logical edges. The simplest case is the Subset edge, which denotes a straightforward, extensional conditonal probability. For instance, it may happen that whenever the Concept for *cat* is present in a system, the Concept for *animal* is as well. Then we would say

```
Subset cat animal
```

(Here we assume a notation where "R A B" denotes an Edge of type R between Vertices A and B.)

On the other hand, it may be that 50% of the time that *cat* is present in the system, *cute* is present as well: then we would say

```
Subset cat cute <.5>
```

where the <.5> denotes the probability, which is a component of the Truth Value associated with the edge. There is a collection of roughly a dozen different logical edge types in SMEPH, which are derived from the Probabilistic Logic Networks framework. We will discuss some of these types in more depth in a later section, in the context of OCP's closely related usage of PLN.

Next, the most basic functional edge is the Execution edge, which is ternary and denotes a relation between a Schema, its input and its output, e.g.

```
Execution father_of Ben_Goertzel Ted_Goertzel
```

for a schema father_of that outputs the father of its argument.

The ExecutionOutput (ExOut) edge denotes the output of a Schema in an implicit way, e.g.

```
ExOut say_hello
```

refers to a particular act of saying hello, whereas

```
ExOut add_numbers {3, 4)
```

refers to the Concept corresponding to 7. Note that this latter example involves a set of three entities: sets are also part of the basic SMEPH knowledge representation. A set may be thought of as a hypergraph edge that points to all its members.

In this manner we may define a set of edges and vertices modeling the habitual activity patterns of a system when in different situations. This is called the *derived hypergraph* of the system. Note that this hypergraph can in principle be constructed no matter what happens inside the system: whether it's a human brain, a formal

neural network, Cyc, OCP, a quantum computer, etc. Of course, constructing the hypergraph in practice is quite a different story: for instance, we currently have no accurate way of measuring the habitual activity patterns inside the human brain. fMRI and PET technologies give only a crude view, though they are continually improving.

Pattern theory enters more deeply here when one thoroughly fleshes out the Inheritance concept. Philosophers of logic have extensively debated the relationship between *extensional* inheritance (inheritance between sets based on their members) and *intensional* inheritance (inheritance between entity-types based on their properties). A variety of formal mechanisms have been proposed to capture this conceptual distinction; see (Wang, 2006, 1995) for a review along with a novel approach utilizing uncertain term logic. Pattern theory provides a novel approach to defining intension: one may associate with each ConceptVertex in a system's derived hypergraph the set of patterns associated with the structural pattern underlying that ConceptVertex. Then, one can define the strength of the IntensionalInheritanceEdge between two ConceptVertices A and B as the percentage of A's pattern-set that is also contained in B's pattern-set. According to this approach, for instance, one could have

```
IntInhEdge whale fish <0.6>

ExtInhEdge whale fish <0.0>
```

since the fish and whale sets have common properties but no common members.

Patternist philosophy comes in here and makes some definite hypotheses about the structure of derived hypergraphs. It suggests that derived hypergraphs should have a dual network structure, and that in highly intelligent systems they should have subgraphs that constitute models of the whole hypergraph (these are *self systems*). SMEPH does not add anything to the patternist view on a philosophical level, but it gives a concrete instantiation to the general ideas of patternism.

Probabilistic and Evolutionary Dynamics

The logical edges in a SMEPH hypergraph are weighted with probabilities, as in the simple example given above. The functional edges may be probabilistically weighted as well, since some Schema may give certain results only some of the time. These probabilities are critical in terms of SMEPH's model of system dynamics; they underly one of SMEPH's three key principles of the dynamics of intelligence,

Principle of Implicit Probabilistic Inference: In an intelligent system, the temporal evolution of the probabilities on the edges in the system's derived hypergraph should approximately obey the rules of probability theory.

The basic idea is that, even if a system's underlying dynamics has no explicit connection to probability theory, nevertheless it must behave roughly as if it does, if it is going to be intelligent. The *roughly* part is important here — it's well known that humans are not terribly accurate in explicitly carrying out formal probabilistic inferences. And yet, in practical contexts where they have experience, humans can make quite accurate judgments — which is all that's required by the above principle, since it's the contexts where experience has occurred that will make up a system's derived hypergraph.

The next key dynamical principle of SMEPH is evolutionary, and states

Principle of Implicit Evolution: In an intelligent system, new Schema and Concepts will continually be created, and the Schema and Concepts that are more useful for achieving system goals (as demonstrated via probabilistic implication of goal achievement) will tend to survive longer.

Note that this principle can be fulfilled in many different ways. The important thing is that system goals are allowed to serve as a selective force.

Another SMEPH dynamical principle pertains to a shorter time-scale than evolution, and states

Principle of Attention Allocation: In an intelligent system, Schema and Concepts that are more useful for attaining short-term goals will tend to consume more of the system's energy. (The balance of attention oriented toward goals pertaining to different time scales will vary from system to system.)

Next, there is the

Principle of Autopoesis: In an intelligent system, if one removes some part of the system and then allows the system's natural dynamics to keep going, a decent approximation to that removed part will often be spontaneously reconstituted.

And the

***Cognitive Equation* Principle**: In an intelligent system, many abstract patterns that are present in the system at a certain time as patterns among other Schema and Concepts, will at a near-future time be present in the system as patterns among elementary system components.

The *Cognitive Equation* Principle, named after the *Cognitive Equation* from the 1994 book *Chaotic Logic*, basically means that Concepts and Schema emergent in the system are *recognized by the system* and then embodied as elementary items in the system — so that patterns among them in their emergent form become, with the passage of time, patterns among them in their directly-system-embodied form. This is a natural consequence of the way intelligent systems continually recognize patterns in themselves.

Note that derived hypergraphs may be constructed corresponding to any complex system which demonstrates a variety of internal dynamical patterns depending on its situation. However, if a system is not intelligent, the evolution of its derived hypergraph can't be expected to follow the above principles.

These principles follow from the psynet model of mind, but they are more precise than the psynet model can be, because they assume a particular formalism for representing the contents of a mind (SMEPH hypergraphs). Of course, no particular mind will be completely described by this sort of hypergraph model; the idea is that this level of approximate description is good enough for many purposes.

The relationship between the human brain/mind and OCP may be explored in a SMEPH context, by considering that both OCP and the human mind can be modeled as SMEPH hypergraphs that obey the principles of implicit probabilistic inference and evolution. Below we will use this approach to help organize our discussion of various concrete results from the cognitive sciences and their relevance for OCP and AGI in general.

From SMEPH to OCP

While SMEPH is a general approach to modeling any intelligent system, it is also possible to create intelligent systems bearing a special relationship to SMEPH. OCP falls into this category, as described on the page FromSMEPHToOCP, as did its predecessor systems Webmind and the NCE. This special relationship makes it particularly easy to analyze these AI systems in SMEPH terms, but it also gives rise to potential confusions.

OpenCogPrime:FromSMEPHToOCP

The SMEPH framework can be used to describe any intelligent system, including any AI system or a human brain. However, it can also be used as a guide for constructing particular AGI systems, such as OpenCogPrime.

OCP represents knowledge internally using a hypergraph data structure that involves nodes and links similar to SMEPH's edges and vertices. However, OCP's vocabulary of node and link types is richer than SMEPH's, and the semantics of its nodes and links are different than that of SMEPH's edges and vertices. For instance, OCP has node types called ConceptNode and SchemaNode, but also others like PredicateNode and various types of PerceptNodes. A OCP ConceptNode will not generally represent a SMEPH Concept edge, because it's rare that OCP's response to a situation will consist solely of activating a single ConceptNode. Rather, the Concept edges in the derived hypergraph of a OCP system will generally correspond to fuzzy sets of OCP nodes and links.

The term Map is used in OCP to refer to a fuzzy set of nodes and links that corresponds to a SMEPH concept or schema; and there is a typology of OCP maps, to be briefly discussed below. Often it happens that a particular OCP node will serve as the *center* of a map, so that e.g. the Concept edge denoting *cat* will consist of a number of nodes and links roughly centered around a ConceptNode that is linked to the WordNode *cat.* But this is not guaranteed — some OCP maps are more diffuse than this with no particular center.

Somewhat similarly, the key SMEPH dynamics are represented explicitly in OCP: probabilistic reasoning is carried out via explicit application of PLN on the OCP hypergraph, evolutionary learning is carried out via application of the BOA optimization algorithm, and attention allocation is carried out via a combination of inference and evolutionary pattern mining. But the SMEPH dynamics also occur implicitly in OCP: emergent maps are reasoned on probabilistically as an indirect consequence of node-and-link level PLN activity; maps evolve as a consequence of the coordinated whole of OCP dynamics; and attention shifts between maps according to complex emergent dynamics.

A SMEPH-based intelligence such as a OCP system will also have a derived hypergraph, which will not be identical to the hypergraph it uses for explicit knowledge representation. However, an interesting feedback loop arises here, in that the intelligence's self-study will generally lead it to recognize large portions of its derived hypergraph as patterns in itself, and then embody these patterns within its concretely implemented knowledge hypergraph. This is closely related to the Cognitive Equation phenomenon described on the SMEPH page, in which an intelligent system continually recognizes patterns in itself and embodies these patterns in its own basic structure (so that new patterns may more easily emerge from them).

Chapter Two: Overview of the OpenCog Prime System

Why Might OpenCog Prime Succeed As an AGI?

1. It is based on a well-reasoned, comprehensive theory of mind, conceptually outlined in *The Hidden Pattern*, which dictates a unified approach to the five key aspects mentioned above
    1. knowledge representation
    2. learning, reasoning and creativity, collectively grouped as knowledge creation
    3. cognitive architecture
    4. embodied, experiential learning
    5. emergent structures and dynamics
2. The specific algorithms and data structures chosen to implement this theory of mind are efficient, robust and scalable and, so is the software implementation
3. Virtual world technology provides a powerful arena for instructing and experimenting with AGI systems
4. The open source methodology allows a vast amount of global brainpower to be brought to bear on working out the numerous details still only partially resolved

Explicit Versus Implicit Knowledge Representation

OCP's knowledge representation must be considered on two levels: implicit and explicit.

The explicit knowledge representation may be viewed as a SMEPH-style generalized hypergraph, which will generally be referred to in these pages as a hypergraph of Nodes and Links, to distinguish it from the Vertices and Edges in the SMEPH derived hypergraph of a OCP system. This includes ConceptNodes and SchemaNodes, where SchemaNodes are represented as mathematical objects using arithmetic, logical and combinatory operators to combine elementary data types and OCP Nodes and Links, designed to enable compact expression of useful cognitive procedures. It also includes a number of other node types including PredicateNodes (SchemaNodes that produce truth values as their outputs) and various kinds of Nodes representing particular kinds of concrete information, such as NumberNodes, WordNodes, PolygonNodes, and so forth.

More information on the explicit KR of OCP may be found on the OpenCogPrime:KnowledgeRepresentation page.

In addition to explicit knowledge representation in terms of Nodes and Links, OCP also incorporates implicit knowledge representation in the form of what are called Maps: collections of Nodes and Links that tend to be utilized together within cognitive processes.

To see the need for maps, consider that even a Node that has a particular meaning attached to it — like the *Iraq* Node, say — doesn't contain much of the meaning of *Iraq* in it. The meaning of *Iraq* lies in the Links attached to this Node, and the Links attached to their Nodes — and the other Nodes and Links not explicitly represented in the system, which will be created by OCP's cognitive algorithms based on the explicitly existent Nodes and Links related to the *Iraq* Node.

This halo of Atoms related to the *Iraq* node is called the *Iraq* map. In general, some maps will center around a particular Atom, like this *Iraq* map, others may not have any particular identifiable center. OCP's cognitive

processes act directly on the level of Nodes and Links, but they must be analyzed in terms of their impact on maps as well. In SMEPH terms, OCP maps may be said to correspond to SMEPH ConceptVertices, and for instance bundles of Links between the Nodes belonging to a map may correspond to a SMEPH Edge between two ConceptVertices

The standard PLN link types such as OpenCogPrime:ExtInhLinks and OpenCogPrime:IntensionalInheritanceLinks exist in OCP as well as SMEPH; these are described in depth in the Probabilistic Logic Networks book. The mathematics of PLN contains many subtleties, and there are relations to prior approaches to uncertain inference including NARS (Wang, 2006, 1995) and Walley's theory of interval probabilities (1991). An essentially complete software implementation of PLN exists within the current OCP codebase and has been tested on various examples of mathematical and commonsense inference.

OCP Atoms (Nodes and Links) are quantified with truth values that, in their simplest form, have two components, one representing probability (*strength*) and the other representing *weight of evidence*; and also with *attention values* that have two components, short-term and long-term importance, representing the estimated value of the Atom on immediate and long-term time-scales.

In practice many Atoms are labeled with OpenCogPrime:CompositeTruthValues rather than elementary ones. A composite truth value contains many component truth values, representing truth values of the Atom in different contexts and according to different estimators.

The vocabulary of node and link types used to represent knowledge in OCP is more comprehensively inventoried at OpenCogPrime:AtomTypes. An incomplete list of node types is as follows:

- OpenCogPrime:ConceptNode
  - *tokens* for links to attach to
- OpenCogPrime:PredicateNode
- OpenCogPrime:ProcedureNode
- OpenCogPrime:PerceptNode
  - Visual, acoustic percepts, etc.
- OpenCogPrime:NumberNode

And an incomplete list of link types is:

- OpenCogPrime:LogicalLink
  - OpenCogPrime:InheritanceLink
  - OpenCogPrime:SimilarityLink
- OpenCogPrime:ImplicationLink
- OpenCogPrime:EquivalenceLink
- OpenCogPrime:IntensionalLogicalRelationship
- OpenCogPrime:HebbianLinks
- OpenCogPrime:ProcedureEvaluationLink

The semantics of these types has been discussed informally in various published overview papers. The logical links are described in depth in the PLN manuscript, and the nonlogical links will be described within pages of this wiki as they are needed.

It is important to note that the OCP declarative knowledge representation is neither a neural net nor a semantic net, though it does have some commonalities with each of these traditional representations. It is not a neural net because it has no activation values, and involves no attempts at low-level brain modeling. However, *attention values* are very loosely analogous to time-averages of neural net activations. On the other hand, it is not a semantic net because of the broad scope of the Atoms in the network (see Figure 5): for example, Atoms may represent percepts, procedures, or parts of concepts. Most OCP Atoms have no corresponding English label. However, most OCP Atoms do have probabilistic truth values, allowing logical semantics.

Knowledge Creation in OpenCog Prime

Learning, reasoning, invention and creativity are all aspects of **knowledge creation** New knowledge is nearly always created via judicious combination and variation of old knowledge — but due to the phenomenon of "emergence," this can lead to the impression of radical novelty Superior capability for knowledge creation is the main thing that separates humans from other animals Knowledge creation allows the creation of context-specific knowledge representations using the basic representational mechanisms available

The OpenCog Prime approach to knowledge creation involves, firstly, positing distinct knowledge creation mechanisms corresponding to the 4 major subtypes of knowledge

1. Sensory: Memory-driven simulation of sensory memory
2. Declarative:
    1. Probabilistic logical inference
    2. Probabilistic Logic Networks formalism
    3. Clustering
    4. Conceptual Blending
    5. Statistical Pattern Mining
3. Procedural: Probabilistic Evolutionary Program Learning
    1. MOSES, PLEASURE algorithms
4. Episodic: Internal simulation
    1. Third Life simulation world

Algorithms for Procedural Knowledge Creation

The key algorithm for procedural knowledge creation used in the OCP design is the MOSES Probabilistic Evolutionary Learning algorithm. MOSES combines the power of two leading AI paradigms: evolutionary and probabilistic learning. As well as its use in AGI, MOSES has a successful track record in bioinformatics, text and data mining, and virtual agent control.

An alternative approach is also being explored, which is complementary rather than contradictory to MOSES: The PLEASURE Algorithm

Algorithms for Declarative Knowledge Creation

The most complex algorithm for declarative knowledge creation in OCP is the Probabilistic Logic networks engine, OpenCogPrime:ProbabilisticLogicNetworks

- The first general, practical integration of probability theory and symbolic logic.
- Extremely broad applicability. Successful track record in bio text mining, virtual agent control.
- Based on mathematics described in Probabilistic Logic Networks, published by Springer in 2008
- Grounding of natural language constructs is provided via inferential integration of data gathered from linguistic and perceptual inputs

XX insert figure KnowCreate1

In addition to PLN, OCP contains multiple heuristics for Atom creation, including "blending" of existing Atoms

XX insert figure KnowCreate 2(blending)

and clustering and other heuristics, see OpenCogPrime:SpeculativeConceptFormation.

Declarative Knowledge Creation via Natural Language Processing

See page on OpenCogPrime:NLP

General Cognitive Dynamics

The knowledge creation mechanisms corresponding to the four knowledge types may be subsumed under a single, universal mathematical scheme of "iterated cognitive transformation"

insert figure KnowCreate3 (forward combo picture)

For a detailed discussion of this general perspective, see http://www.goertzel.org/dynapsyc/2006/ForwardBackward.htm, and the wiki topics MindOntology:FocusedCognitiveProcess

However, none of the knowledge-subtype-specific knowledge creation mechanisms can stand on its own, except for simple or highly specialized problems.

To support general intelligence, the four basic knowledge creation mechanisms must richly interact, and support each other. This interaction must occur within each cognitive unit in the overall Cognitive Architecture

In computer science language, we may say that each of these mechanisms is subject to combinatorial explosions, and must be interconnected in such a way that they can help each other with pruning. The understanding of the KC mechanisms as manifestations of the same universal cognitive dynamic, aids in working out the details of these interactions.

Attention Allocation

Regulating all this knowledge creation across a large knowledge base requires robust mechanisms for OpenCogPrime:AttentionAllocation — allocation of both processing and memory

The allocation of attention based on identified patterns of goal-achievement is known as www.goertzel.org/dynapsyc/2006/ForwardBackward.htm

Attention allocation is itself a subtle AI problem integrating declarative, procedural and episodic knowledge and knowledge creation

The OpenCogPrime approach to attention allocation involves artificial economics, with two separate currencies:

1. STI (short-term importance) currency, corresponding to processor time
2. LTI (long-term importance) currency, corresponding to RAM

Each Atom in the AtomTable has an AttentionValue consisting of OpenCogPrime:STI and OpenCogPrime:LTI currency values, along with (in most cases) a probabilistic truth value

The following figure illustrates the semantics of STI and LTI

insert figure KnowCreate4 XX

Each node or link in the NCE's knowledge network is tagged with a probabilistic truth value, and also with an "attention value", containing Short-Term Importance and Long-Term Importance components.

An artificial-economics-based process is used to update these attention values dynamically — a complex, adaptive nonlinear process.

Attention Allocation & Knowledge Creation

Patterns among currency values may be used as raw material for knowledge creation, a process called OpenCogPrime:MapFormation

Aspects of this process, corresponding to the four key types of knowledge, include:

1. Declarative map formation: Formation of a new concept grouping together concepts that have often been active together (often had high STI at the same time) … a clustering problem
2. Procedural map formation: Formation of a new procedure whose execution is predicted to generate a time series of STI values similar to one frequently historically observed … a procedure learning problem
3. Episodic map formation: Formation of new episodic memories with the property that experiencing/remembering these episodes would generate a time-series of STI values similar to one frequently historically observed … an optimization problem
4. Sensory map formation: Formation of new sensory memories with the property that perceiving these sensations would generate a pattern of STI values similar to one frequently historically observed

The following diagram illustrates the map formation process. Atoms associated in a dynamic "map" may be grouped to form new Atoms: the Atomspace hence explicitly representing patterns in itself

insert figure KnowCreate5 XX

Summary of Some Synergies Between Knowledge Creation Processes

A detailed discussion of the synergies between the different knowledge creation processes in OpenCog Prime may be found at OpenCogPrime:EssentialSynergies. The next few paragraphs, on this page, just give a brief overview of the topic.

How Declarative KC helps Procedural KC

MOSES and PLEASURE both require

1. Procedure normalization, which involves execution of logical rules, which for the case of complex programmatic constructs, may require nontrivial logical inference
2. Probabilistic modeling of the factors distinguishing good from bad procedures (for a certain purpose), which may benefit from the capability of advanced probabilistic inference to incorporate diverse historical factors

How Procedural KC helps Declarative KC

State-of-the-art logical reasoning engines (probabilistic or not) falter when it comes to "inference control" of complex inferences, or inferences over large knowledge bases At any given point in a chain of reasoning, they know which inference steps are correct to take, but not which ones are potentially useful at achieving the overall inference goal When logical inference gets stuck in reasoning about some concept, one recourse is to use procedure learning to figure out new procedural rules for distinguishing that concept from others These rules may then be fed into the inference process, adding new information that often allows greater-confidence inference control

How Episodic KC helps Declarative KC

When an inference process can't tell which of many possible directions to choose, another potential solution is to rely on life-history: on the memory-store of prior inferences done in related situations.

In inference control as elsewhere, what worked in the past is often a decent guide to what will work in the future.

How Declarative KC helps Episodic KC

An embodied organism experiences a great number of episodes during its lifetime, and without some kind of abstract organization, it will only be able to call these to mind via simple associative cueing. Declarative knowledge creation, acting on the episodic memory store, forms ontologies of episodes, allowing memory access based on abstract as well as associative cues. Furthermore, episodes are not generally stored in the memory in complete detail. Declarative knowledge is used to fill in the gaps — an aspect of "the constructive nature of memory".

How Procedural KC helps Episodic KC

Suppose the mind wants to re-create an episode that it doesn't recall in full detail, or to create an episode that it never actually experienced?

In many cases this episode will involve other agents with their own dynamic behaviors

Procedure-learning mechanisms are used to infer the processes governing other agents' behaviors, which is needed to simulate other agents

How Episodic KC helps Procedural KC

MOSES and PLEASURE require probabilistic modeling of the factors distinguishing good from bad procedures (for a certain purpose), which may benefit from simple associative priming based on episodic memory of what procedures worked before in similar circumstances

Interactions with Sensory KC

The three other forms of KC benefit from sensory KC simply via acting on the input provided by sensory KC, in the same manner as they act on direct sensory input And, sensory KC works primarily via having the sensory cognitive units stimulated by other cognitive units generating "mock stimuli" e.g.

1. Daydreaming fake sensations and experiences one never had (episodic)
2. Using imagistic thought to help guide mathematical reasoning (declerative/procedural)

OpenCog Prime Cognitive Architecture

Cognitive architecture is the aspect of AGI design that seeks to answer questions such as:

1. What parts should a mind-system be decomposed into?
2. How should these parts relate to each other?
3. How should these parts work together to cause the system to carry out purposeful actions in the world?

Division into High-Level Functional Units

In the OpenCog Prime design, cognitive architecture is based on conceiving the mind as divided into multiple cognitive units, where each unit

1. focuses on a certain type of information or a certain way of allocation attention to information
2. processes all the 4 major types of knowledge in a synthetic way

The most important cognitive units handle:

1. Sensation in various modalities
2. Action
3. General "background" cognition
4. Focused cognition
5. Language
6. Social modeling (self and other)

At a high level OpenCog Prime's cognitive architecture is based on the state of the art in cognitive psychology and cognitive neuroscience. Most cognitive functions are distributed across the whole system, yet principally guided by some particular module. The following is a high level architecture diagram for OpenCog Prime:

CogArch1 - insert diagram XXX

It doesn't really tell you much ... the meat of the design is in what lies inside the boxes, and how the insides of the boxes dynamically interact with each other and give rise to system-wide attractors and other emergent dynamic phenomena.

Tying Cognitive Architecture in with Software Architecture

Each of the high-level cognitive units in the above diagram, may be implemented on the software level as a OpenCog Unit that consists of potentially multiple machines sharing a common AtomSpace, as shown in the following figure:

CogArch 2 - insert diagram XXX

In turn, each machine in each of these Units consists of an AtomTable acted on by multiple MindAgents, with interactions mediated by a Scheduler, as shown in the following figure:

CogArch 3 - insert diagram XXX

Goal-Driven Cognitive Dynamics

The combined action of OpenCog Prime's functional sub-units follows the basic logic of animal behavior:

Enact a procedure so that

Context & Procedure ==> Goals

i.e. at each moment, based on its observations and memories, the system chooses to enact procedures that it estimates (based on the properties of the current context) will enable it to achieve its goals, over the time-scales these goals refer to.

For the details of this process, see the page on OpenCogPrime Psyche.

The "procedures" in the above paragraphs may be either Procedure objects or more complex procedures formed by the coordinated enaction of multiple Procedure objects.

Explicit versus Implicit Goals

There is an important distinction between explicit goals and implicit goals

1. Explicit goals: the objective-functions the system explicitly chooses actions in order to maximize, which are represented in Nodes that are explicitly marked as Goals. # Implicit goals: the objective-functions the system actually does habitually maximize, in practice

For a system that is both rational, and capable with respect to its goals in its environment, these will be basically the same. But in many real cases, they may be radically different

Not all of OpenCog's dynamics are explicitly guided by the pursuit of explicitly defined goals. This is largely for computational efficiency reasons: trying to

Goal Dynamics

A few key points about goals are as follows:

1. A sufficiently intelligent system is continually creating new subgoals of its current goals, a process called OpenCogPrime:Subgoal Refinement
2. Some intelligent systems may be able to replace their Top-Level Supergoals with new ones, based on various dynamics
3. Goals may operate on radically different time-scales
4. Humans habitually experience "subgoal alienation" -- what was once a subgoal of some other goal, becomes a top-level goal in itself. AI's need not be so prone to this phenomenon

Implementation Overview

The cognitive architecture within which the representational, learning and reasoning mechanisms discussed above exist within OCP is a fairly simple one. A OCP instance is divided into a set of Units, each of which contains an AtomTable containing a hypergraph of Nodes and Links, and also a set of MindAgent objects embodying various cognitive processes. The MindAgents are perpetually cycled through, carrying out recurrent actions and creating Task objects that carry out processor-intensive one-time actions. Different Units deal with different high-level cognitive functions and may contain different mixes of MindAgents, or at least differently-tuned MindAgents. Conceptually, the idea is that each Unit uses the same knowledge representations and cognitive mechanisms to achieve a particular aspect of intelligence, such as perception, language learning, abstract cognition, action selection, procedure learning, etc.

Note that a Unit may span several machines or may be localized on a single machine: in the multi-machine case, Nodes on one machine may link to Nodes living in other machines within the same unit. On the other hand, Atoms in one Unit may not directly link to Atoms in another Unit; though different Units may of course transport Atoms amongst each other. This architecture is workable to the extent that Units may be defined corresponding to pragmatically distinct areas of mental function, e.g. a Unit for language processing, a Unit for visual perception, etc.

The full cognitive architecture has not yet been implemented, but an architecture capable of supporting it is in place, along with various key components. The following diagram illustrates an *Experiential Learning Focused OCP Configuration*:

XXX insert diagram

Many different OCP configurations are possible within the overall design, but our plan is to work with a specific OCP configuration, intended for *experiential learning* and more specifically, for a OCP system that controls a real or simulated body that is perceiving and acting in some world. The breakdown into units conceived for this purpose, roughly indicated in Figure 8, is based loosely on ideas from cognitive science, and is fairly similar to that presented in other integrative AGI architectures proposed by Stan Franklin (2006), Aaron Sloman (1999) and others. However, the specifics of the breakdown have been chosen with care, with a view toward ensuring that the coordinated dynamics of the mechanisms and units will be able to give rise to the emergent structures and dynamics associated with intelligence, including a sophisticated self-model and an appropriately adaptive *moving focus of attention.*

Currently we are not working with physical robotics but are rather using OCP to control a simple simulated body in a 3D simulation world called AGISim (Goertzel et al, 2006). It would also be possible to construct OCP configurations unrelated to any kind of embodiment; for instance, we have designed a configuration intended specifically for mathematical theorem-proving. However, as argued in The Hidden Pattern, we believe that pursuing some form of embodiment is likely the best way to approach AGI. This is not because intelligence intrinsically requires embodiment, but rather because physical environments present a host of useful cognitive problems at various levels of complexity, and also because understanding of human beings and human language will probably be much easier for AI's that share humans' grounding in physical environments.

OCP's experiential learning configuration centers around a Unit called the Central Active Memory, which is the primary cognitive engine of the system. There is also a Unit called the Global Attentional Focus, which deals with Atoms that have been judged particularly important and subjects them to intensive cognitive processing. There are Units dealing with sensory processing and motor control; and then Units dealing with highly intensive PLN or PEL based pattern recognition, using control mechanisms that are not friendly about ceding processor time to other cognitive processes. Each Unit may potentially span multiple machines; the idea is that communication within a Unit must be very rapid, whereas communication among Units may be slower.

The focus on experiential learning leads to yet another way of categorizing the OCP system's cognitive activities: goal-driven versus ambient. This classification is orthogonal to the control/forward/backward trichotomy discussed above. Ambient cognitive activity includes for instance

- MindAgents that carry out basic PLN operations on the AtomTable, deriving obvious conclusions from existing knowledge
- MindAgents that carry out basic perceptual activity, e.g. recognizing coherent objects in the perceptual stimuli coming into the system
- MindAgents related to attention allocation and assignment of credit
- MindAgents involved in moving Atoms between disk and RAM.

Goal-driven activity, on the other hand, involves an explicitly maintained list of goals that is stored in the Global Attentional Focus and Central Active Memory. Two key processes are involved:

- Learning SchemaNodes that, if activated, are expected to lead to goal achievement
- Activating SchemaNodes that, if activated, are expected to lead to goal achievement
- The goal-driven learning process is ultimately a form of *backward-chaining learning,* but subtler than usual backward chaining due to its interweaving of PLN and PEL and its reliance on multiple cognitive Units.

Emergence in OpenCog Prime

The Atoms, MindAgents and Units within OCP are critical to the system's intended intelligence — but they are critical as *means to an end.* The essence of the OCP system's mind, if the system operates as intended, will be the emergent dynamical structures arising in the Atom-network as it evolves over time. These, not the specific software structures *wired into* the system by its designers and programmers, will be the stuff of the OCP system's *mind* as subjectively self-perceived.

As an important example, one of the key things we hope to see via teaching OCP in the AGISim environment is the adaptive emergence within the system's knowledge base of an active and effectively evolving *phenomenal self.* The process of the emergence of the self may, we hypothesize, be productively modeled in terms of the processes of forward and backward synthesis discussed above. This point is made carefully in (Goertzel, 2006) and just briefly summarized here.

What is ventured there is that the dynamic pattern of alternating forward and backward synthesis may play a fundamental role in cognition. Put simply, forward synthesis creates new mental forms by combining existing ones. Then, backward synthesis seeks simple explanations for the forms in the mind, including the newly created ones; and, this explanation itself then comprises additional new forms in the mind, to be used as fodder for the next round of forward synthesis. Or, to put it yet more simply:

… Combine … Explain … Combine … Explain … Combine …

This sort of dynamic may be expressed formally, in a OCP context, as a dynamical iteration on the space of Atoms. One may then speak about attractors of this iteration: fixed points, limit cycles and strange attractors. And one may hypothesize some key emergent cognitive structures are strange attractors of this equation. I.e., the iterative dynamic of combination and explanation leads to the emergence of certain complex structures that are, in essence, maintained when one recombines their parts and then seeks to explain the recombinations. These structures are built in the first place through iterative recombination and explanation, and then survive in the mind because they are conserved by this process. They then ongoingly guide the construction and destruction of various other temporary mental structures that are not so conserved. Specifically, we suggest that both self and attentional focus may be viewed as strange attractors of this iteration. Here we will focus only on self.

The *self* in this context refers to the *phenomenal self* (Metzinger, 2004) or *self-model.* That is, the self is the model that a system builds internally, reflecting the patterns observed in the (external and internal) world that directly pertain to the system itself. As is well known in everyday human life, self-models need not be completely accurate to be useful; and in the presence of certain psychological factors, a more accurate self-model may not necessarily be advantageous. But a self-model that is too badly inaccurate will lead to a badly-functioning system that is unable to effectively act toward the achievement of its own goals.

The value of a self-model for any intelligent system carrying out embodied agentive cognition is obvious. And beyond this, another primary use of the self is as a foundation for metaphors and analogies in various domains. A self-model can in many cases form a self-fulfilling prophecy (to make an obvious double-entendre'!). Actions are generated based on one's model of what sorts of actions one can and/or should take; and the results of these actions are then incorporated into one's self-model. If a self-model proves a generally bad guide to action

selection, this may never be discovered, unless said self-model includes the knowledge that semi-random experimentation is often useful.

In what sense, then, may it be said that self is an attractor of iterated forward-backward synthesis? Backward synthesis infers the self from observations of system behavior. The system asks: What kind of system might we be, in order to give rise to these behaviors that we observe myself carrying out? Based on asking itself this question, it constructs a model of itself, i.e. it constructs a self. Then, this self guides the system's behavior: it builds new logical relationships its self-model and various other entities, in order to guide its future actions oriented toward achieving its goals. Based on the behaviors new induced via this constructive, forward-synthesis activity, the system may then engage in backward synthesis again and ask: What must we be now, in order to have carried out these new actions? And so on.

Our hypothesis is that after repeated iterations of this sort, in infancy, finally during early childhood a kind of self-reinforcing attractor occurs, and we have a self-model that is resilient and doesn't change dramatically when new instances of action- or explanation-generation occur. This is not strictly a mathematical attractor, though, because over a long period of time the self may well shift significantly. But, for a mature self, many hundreds of thousands or millions of forward-backward synthesis cycles may occur before the self-model is dramatically modified. For relatively long periods of time, small changes within the context of the existing self may suffice to allow the system to control itself intelligently.

And of course the phenomenal self is not the only structure that must emerge in this sort of way, if OCP is to give rise to powerful AGI. Another example of an emergent structure is the OpenCogPrime:AttentionalFocus — the "moving bubble of attention" that constitutes the system's focus of conscious attention at a point in time. Another example is the MindOntology:DualNetwork, consisting of harmonized and synchronized hierarchical and heterarchical patterns of activity. All these structures must emerge and self-perpetuate, grow and mature based on the coordinated activity of MindAgents on Atom-networks situated in Units, if OCP is to make the leap from being a software system to being an autonomous and reflective intelligence.

Chapter Three: Atoms, Atomspaces and So Forth

A Formalization of the Atom Concept

This page presents one among many possible mathematical formalizations of the Atom concept as used in OCP. Furthermore it's not presented in a totally mathematically rigorous style. However it's presented here in case it may provide someone with useful clarification of the concept.

Spaces

In abstract discussions on this page and others, we'll often use the word *space* in a generic sense. All *spaces* mentioned here are finite in any particular example system; the reader, if not inclined toward a finitist philosophy of mathematics, may consider them as countably infinite.

Sometimes in defining spaces we will use the Union_k operator, which should be understood as shorthand for k=0,1,2,, K for some relevant large K (sometimes the K is specified, sometimes not).

Finally, we will often have need here to refer to function spaces. Where A and B are two spaces, we will use the notation [A → B] to denote the space of functions mapping A into B. This notation may be nested, e.g. [[A → B] → C] denotes the space of functions that map *functions mapping A into B* into C.

Weighted, Labeled Hypergraphs

In this section we introduce the formalism of weighted, labeled hypergraphs on which both OCP and SMEPH are based. The *hypergraph* concept in mathematics is a fairly broad one — generically the term is used to refer to graphs (of the node-and-link kind, not the curve-in-a-space kind) in which edges may join multiple vertices (Bollobas, 1998). The weighted, labeled hypergraphs we use here have some peculiarities, which we'll describe here, but they do fit within the general purvey of mathematical hypergraph theory.

Our formalization begins with the concept of an Atom. Atoms are the basic entities of which a hypergraph is composed. Both edges and vertices are to be considered as particular types of Atoms.

Atoms are *weighted*, which means formally that they are objects that are associated with sub-objects called Values, where we define a Value as a finite structure that is either a basic value object, or an ordered list of Values. The basic value objects we're working with at the moment are characters, integers and floating-point numbers. There are different types of Values, exemplified by TruthValue and AttentionValue, the two Value types that all Atoms mandatorily contain.

Each Value type comes along with a *get* mapping that maps each Atom into the Value of that type that it contains; these *get* mappings are called Valuators and live in the space

```
Valuator = [Atom → Val]
```

So for instance the Valuator called GetTruthValue is defined by

```
GetTruthValue (A) = A.TruthValue
```

The next basic concept we need to introduce is that of a tuple. Tuples may be ordered or unordered. The space of Atoms may be partitioned into two initial categories: Atoms that are tuples, and Atoms that are not. As noted above, in a SMEPH *derived hypergraph,* the Atoms that are tuples we will call Edges, and the other ones we will call Vertices. On the other hand, in OCP's concrete knowledge representation, we will refer to Links and Nodes instead.

As noted above, in the particular case of the hypergraph explicitly used for knowledge representation in OCP, we will shift to the alternate terminology (Node, Link) instead of (Edge, Vertex). We will retain the latter generic terms when discussing the *derived hypergraphs* associated with a complex system such as OCP.

Summing up so far, we have defined the spaces:

- Atom — the space of all Atoms
- Vertex — the space of all Vertices
- Edge — the space of all Edges
- Atom = Edge Vertex
- Unordered-Edge = Union_k (the set of unordered k-tuples of elements of Atom)

- Ordered-Edge = Union_k (the set of ordered k-tuples of elements of Atom)
- Edge = (Ordered-Edge Union (Unordered-Edge)

We have mentioned TruthValue and AttentionValue objects. These are described by defining valuator functions:

```
Truth-Valuator = [ Atom → Truth-Value ]

Attention-Valuator = [ Atom → Attention-Value ]
```

Furthermore, we will burden our Atoms with additional values.

Firstly, each atom has a type,

```
Atom-Type = the space of atom types

Atom-Type = Node-Type Union Edge-Type
```

Pragmatically, the atom type values we use in OCP are invariably strings, i.e. lists of characters.

Atom types are used in each particular OCP by the definition of particular associated valuator functions:

Type-Valuator = [Atom → Atom-Type]

```
Atom-Type = Vertex-Type  Edge-Type
```

Many of OCP's dynamics are Vertex and Edge type dependent.

In the Introduction we have given a detailed list of the actual Node and Link types used in a real OCP. For now, we will stay a bit more abstract, and look at the general sorts of things we will need to do with Nodes and Links.

In addition to the Type, TruthValue and AttentionValue valuators defined above, there are also valuators associated with particular Vertex types, such as:

```
Boolean-Valuator = [ BooleanNode → Boolean ]

Number-Valuator = [ NumberNode → Number ]

Character-Valuator = [ CharacterNode → Character ]

Time-Valuator = [ Atom → Time]

Schema-Valuator = [ Atom → Schema]

Predicate-Valuator = [ Atom → Predicate]
```

The spaces of Schema and Predicates will be defined a little later.

It may be worth reiterating how our notation works, in the definition of these abstract spaces of valuators. The idea is that, e.g., Schema-Valuator stands for the class of all possible mappings from sets of Atoms into

Schema. Each particular OCP instance, at each point in time, is then associated with a specific SchemaValue encapsulating a mapping that takes Atoms into Schema (this mapping lies in the space called Schema-Valuator). This mapping is not defined on all Atoms on the system; it is only defined on SchemaNodes, they are its domain.

OpenCogPrime:Combinator

One of the tools used in formalizing and implementing OCP Schemata and Predicates is a branch of mathematics called combinatory logic. This section briefly reviews combinatory logic and explains its utilization in OCP schemata and predicates.

Combinatory logic is a fairly obscure branch of mathematics, generally well known only in the functional programming community. However, we strongly believe that it is more than just a mathematical trick; we feel that its basic insight of representing complex structures as *functions of functions of functions* gets at something important for AI.

For the reader who wishes to go deeper on the mathematical side, the best explanation of the basic principles of combinatory logic that we have found is in the functional programming textbook by Field and Harrison (1988). The more ambitious and mathematically-oriented reader is referred to the original source, Combinatory Logic by Haskell Curry and Robert Feys (1958), which is difficult reading but well worth the effort required (at least where Volume 1 is concerned; we have doubts whether anyone has ever completed Volume 2).

Basic Ideas of Combinatory Logic

Conceptually, combinatory logic is simple. One begins with a few basic entities, called combinators. These combinators can take other combinators as inputs, and produce other combinators as outputs. This basic operation of *action* or *function application* is typically denoted by adjacency, e.g.

```
f g
```

refers to the combinator that f will output when given g as an input.

Application is assumed to left-associate (*currying*), so that

```
f g h = (f g) h
```

This convention takes a while to get used to, but provides great simplicity of expression once fully internalized. If one wants right association, one must explicitly insert parentheses, as in

```
f (g h)
```

which feeds as input to f, the output that g gives on being given input h.

The remarkable thing is that, from this simple framework alone, one can generate anything at all. In other words, this is yet another way of providing universal computational capability.

The standard, almost minimal formulation of combinatory logic uses only two combinators, S and K (to be defined just below). These are enough to give the combinatory framework arbitrary computational power.

We will use underlined characters to denote combinators (e.g. S, K), which allows us to use non-underlined letters like S and K to denote other things without possibility of confusion. This underlining is not standard combinatory logic notation.

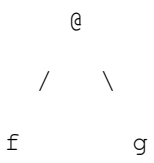
Formalization of Combinators

More formally speaking, one may specify a combinator algebra as follows. One starts with a finite set of combinators,

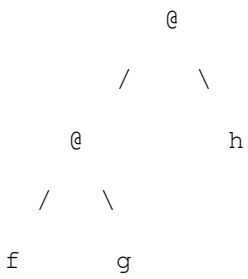
```
Basic-Combinator = { f1 ,…, fn }
```

Then one introduces the notion of a combinator expression, which can be expressed as a (finite) tree of a special kind.

The nodes of a *Combo tree* contain either combinators or @ symbols. The @ symbol denotes function application. So for instance, we may have



which means *f is applied to g*, or in curried notation, f g, or in standard functional notation f(g). Or one may have



which means

- curried: (f g) h
- uncurried: (f(g))(h)

Formally, the space of Combo trees is defined by the fact that

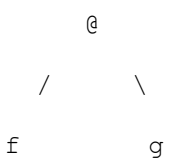
- leaf nodes have to have basic combinators at them, not @ symbols
- non-leaf nodes have to have @ symbols at them

One may then define

```
Combinator-Expressions
```

as the space of combinator expression obtained from one's given set of basic combinators.

The essence of combinator activity lies in the *multiplication table* associated with a set of basic combinators. Each expression of the form



where f and g are basic combinators, either

- evaluates to NULL, meaning that f cannot act on g
- evaluates to some basic combinator

This allows us to define an evaluation map on the space of combinator expressions, where we say that

- a combinator expression evaluates to NULL if any of its subexpressions evaluate to NULL
- otherwise, a combinator expression evaluates to what its root node evaluates to, after recursive evaluation of all subexpressions

In principle, the combinator multiplication table could be entirely random and unstructured. However, this case is not all that interesting in practice. Instead, there are usually patterns in evaluations, which can be encapsulated in variable-bearing *macro rules* or *substitution rules.*

The basic combinators called S and K are the only ones needed to give combinatory logic arbitrarily great expressive power. In OCP we make use of a much broader set of combinators, beginning with the slightly expanded combinator set S, K, I, B, C defined as follows:

```
I x = x

K e f  = e

S f g x= (f x) (g x)

B f x y = f (x y)

C f x y = f y x
```

Another important combinator is the Y-combinator

```
Y f = Y ( Y f)
```

but we have avoided this one in OCP because, in practice, it has a high probability of giving rise to nonterminating expressions. We have also introduced a number of higher-level combinators for dealing with things like functions and lists, which will be discussed later on.

It is not hard to figure out how to represent I, B and C in terms of S and K. The representation of Y in these terms is a little trickier, and is given by the definitions

```
U f x ==> f (x x)

Y = S (K (U I)) U
```

Note that adjacency always means function application in combinatory logic. To denote function composition one uses the compositor combinator, B.

In practice one usually wants to extend the basic combinator vocabulary even further. For instance, one can introduce the notion of an ordered or unordered list as a primitive, so that one can then have expressions like

```
f (x, y)
```

The list operator called "fold" defined by

```
fold f v [] = v

fold f v (x : xs) = f x (fold f v xs)
```

can be particularly useful. (This is called foldl in Haskell.)

One can also introduce primitives like numbers, strings, and so forth. In this sense the OCP schema module is much like a functional programming language. The reason one introduces these extra primitives, is that otherwise the expression of some intuitively simple things becomes overly complicated. For instance, in the minimalist approach, using only the basic combinators given above, one represents numbers, numerical operators, and logical operators in terms of a *code* which expresses each of them as a certain combination of S and K combinators. Elegant, but not so pragmatic.

From Combinators to ProcedureNodes

How are Combinators actually used in OCP?

The basic principles are as follows:

- OCP Atoms, besides SchemaNodes, are treated as ìconstantsî that can be acted on by schema
- We introduce a set of ìelementary schema,î including arithmetic and logical operators, and also schema that take actions in the outside world (like sending out a character of text or moving an actuator) and schema that monitor the overall state of the system. The full elementary schema set will be presented in the following Chapter.
- We use the standard combinators {S, K, I, B , C } as defined above to build up complex expressions from the given constants and elementary schema

This is not unlike how combinatory logic is used to build up functional programming languages.

The class

```
ComboTree
```

is then used to refer to the combinator expressions built up from the elementary schema and constants in OCP; and the class

```
Predicate in ComboTree
```

refers to those schema that output TruthValue objects.

SchemaNodes are so named because the mapping

```
Schema-Valuator  in [ Atom --> ComboTree ]
```

associates them with Procedures. Similarly, for PredicateNodes, we have a mapping

```
PredicateValuator  in  [ Atom --> Predicate ]
```

The set of elementary Schemata and Predicates will be the same for a certain OCP system at all points in time -- *unless* one has a OCP system that is modifying its own codebase. Otherwise, new schema and predicate expressions will be created, but using the same basic hard-wired vocabulary. Of course, the given vocabulary of elementary schemata has universal computational power; but even so, it is quite possible that enhancements to the set of elementary schema could make the system more intelligent in certain contexts.

[edit] Variables and Combinators in OCP

Originally, in an early version of the NCE system (which was OCP's inspiration), there was a design intention to eschew variables entirely in procedures and predicates, and exclusively use variable-free combinator expressions to represent complex functions. In spite of innovations like Turner combinators, however, this eventually came to seem unwise due to the computational complexity of expressing certain intuitively simple functions in a variable-free way. So in the current OCP design, the intention is for ComboTrees to be able to utilize both variables and higher-order combinators, thus achieving the same kind of flexibility that exists in modern functional programming languages (which use director strings and other variable-like formalisms to work around the practical limitations of the pure-combinator approach).

[edit] Varieties of Schemata

A list of the various specific ìelementary schemataî currently used in OCP will be given later on. Here, as a preliminary, we make some general comments about the kinds of schema in the system.

The main type of schema associated with SchemaNodes is the atom-transforming schema,

```
Atom-Transforming-Schema  in [Atom --> Atom]
```

The basic combinators (S, K, etc.) fall into this category, for instance, as do Boolean schemata like AND and OR. They are embodied in elementary schema nodes, hence they themselves are represented by Atoms; so their actions on each other may be represented as actions of Atoms on Atoms.

Similarly, PredicateNodes are commonly associated with predicates of the form

```
Atom-Based-Predicate in [Atom --> TruthValue]
```

But there are other schema varieties as well; for instance,

```
Feeling-Predicate in  [ Atomspace --> Truth-Value ]
```

There are two types of schemata which must be conceived of as special kinds of CIM-Dynamics:

```
Atom-Deleting-Schema    in CIM-Dynamic

Atom-Creating-Schema  in  CIM-Dynamic
```

An atom deleting schema takes an Atomspace and removes a particular Atom from it. An atom creating schema takes an Atomspace and puts a particular atom in it.

Atom-creating and atom-deleting schemas are elements of

```
[Atomspace --> Atomspace]
```

Where A and B are two Atomspaces, an atom-creating schema f has the property

```
f A = B implies (A  subset B)
```

whereas an atom-deleting schema f has the property

```
f A = B implies (B  subset A)
```

In the case of multipart or caching Atomspaces, to be introduced later, there may be atom creation schema that specifically place schemata in particular regions of the Atomspace.

Altogether, these are the only kinds of schema in the current OCP system, so we may say formally that

```
Combo tree =

Atom-Transforming-Schema  ''Union''

Atom-Deleting-Schema  ''Union''

Atom-Creating-Schema  ''Union''

Predicate
```

and

```
Predicate = Atom-Based-Predicate  Union Feeling-Predicate
```

The elementary schema in any given OCP may be thought of as a ìschema vocabularyî; and we may say that OCP is supplied with a set of ìschema sub-vocabularies,î consisting of schema carrying out particular sorts of purposes. These sub-vocabularies may be defined as sets

```
Atom-Transforming-Schema-Vocabulary

Atom-Creating-Schema-Vocabulary

Atom-Deleting-Schema-Vocabulary

Predicate-Vocabulary

Truth-Value-Schema-Vocabulary

Feeling-Vocabulary
```

We may then define an overall schema vocabulary as:

```
Schema-Vocabulary = Atom-Transforming-Schema-Vocabulary  ''Union''

Atom-Deleting-Schema-Vocabulary ''Union''

Atom-Creating-Schema-Vocabulary  ''Union''

Feeling-Vocabulary     ''Union''

Atom-Based-Predicate-Vocabulary
```

In general, each OCP is associated with a particular schema vocabulary.

On the implementation level, SchemaNodes and PredicateNodes are associated with combinator expressions formed from the schemata in the systemís schema vocabulary, taking the Atoms in the system as arguments. The combinator expression corresponding to a Node is encapsulated in an object called a ComboTree, which is contained in a structure called the ProcedureRepository (in which a ComboTree is indexed according to the Node that corresponds to it); more details on this will be given in a later chapter.

OpenCogPrime:Map

OCP knowledge representation exists on two levels: the concretely-implemented and the emergent. The primary concept underlying emergent KR in OCP is the map, which is discussed on this page among other places.

OCP explicitly uses a weighted labeled hypergraph as a low-level knowledge representation; these are the Nodes and Links discussed extensively on other pages. But it also has emergent dynamics that may be modeled using a weighted labeled hypergraph, according to the *derived hypergraph* concept presented in the Introduction.

Essentially and intuitively, what we mean by a OCP *map* is a set of OCP Atoms whose STI levels display a high degree of cooperativity. In practice, this cooperativity may manifest itself in many ways. Each OCP map may

be considered as a Vertex in a derived hypergraph associated with a OCP system. Edges in the derived hypergraph then denote relationships between maps. This network of Edges and Vertices is just as important an aspect of OCP intelligence as the explicit, concretely-implemented network of Nodes and Links.

One very common kind of map is the map defined by simultaneous STI. This kind of map is simply a set of Atoms that tend to become active at the same time. For instance, suppose one associates each Atom with a time series defined by that Atom's STI values over time. Then a simultaneous STI map is simply a cluster in the space of STI-time-series. One can also look at clusters in the space of importance-time-series. These two sets of clusters (those defined by STI, and those defined by importance) should be fairly similar, though not identical. These maps appear as ConceptVertices in the derived hypergraph.

Another very common kind of map is the one defined by a series of Atoms that tend to become active in a certain sequence. This is the case, for instance, with schema maps, which are *distributed programs* consisting of sets of SchemaNodes that tend to execute in a certain order, carrying out a coherent overall function. This sort of map is definable as a pattern in the STI time series associated with Atoms, but not as a cluster in STI time series space. Rather, it is a set of probabilistic rules. An example of this sort of rule would be:

```
Atoms A1 and A9 active at time T
AND
Atoms A2 and A5 active at time T+1
AND
Atoms A1 and A8 active at time T+2
```

This is a simple map involving 5 Atoms and 3 points in time. Real examples may be much more complex, of course. This sort of map appears as a SchemaVertex in the derived hypergraph.

Generally speaking, we may say that a map is a set of Atoms whose STI and importance time series, taken as a whole, embody prominent probabilistic rules. A cluster is an example of such a probabilistic rule, as is the kind of temporally-synchronized STI pattern exemplified above.

More rigorously, we believe the most relevant probabilistic rules involved here are what we call *generalized predictive patterns*. Essentially, a generalized predictive pattern in a set of time series is a schema that can approximately predict the value of one of the time series in the set at time T, based on

1. the values of all the time series in the set at the times (T-K,…,T-1)
2. the values of all the time series but itself at time T.

A pure predictive time series is one that predicts the values of one of the time series in the set at time T, based only on the first one of the two factors listed.

Clustering allows one to predict the value of one Atom's STI or importance time series at time T, based on the values at time T of the STI and importance time series associated with other elements of the same cluster. Temporally-synchronized maps, on the other hand, are *pure predictive patterns* which predict the STI levels of certain Atoms based on the past STIs of other related Atoms.

Generally speaking, then, we may define a set of maps associated with a given OCP instance as

Map-Space = Union_k (Atom^k *Union* GeneralizedPredictivePattern)

A map is a set of k Atoms (for some k), combined with a generally predictive pattern involving the Atoms in the set (the predictive pattern tells what is the notable regularity by which the set of Atoms changes over time). The individual map space associated with a particular OCP instance, must be considered as a fuzzy set, where the degree of membership is defined by the (strength * weight-of-evidence) of the generalized predictive pattern involved.

The *derived hypergraph* idea is a mapping from Map-Spaces into weighted labeled hypergraphs, achieved via, as suggested above, treating appropriate tuples of Atoms as ConceptVertices and appropriate GeneralizedPredictivePatterns as SchemaVertices. Edges may then be drawn between these Vertices based on various sorts of observed relationships. For instance (to introduce some PLN), if ConceptVertex A is observed 80% of the time that ConceptVertex B is observed, then we might say

ExtensionalInheritanceEdge A B <.8>

Maps may drive OCP dynamics implicitly, without OCP or any human observer identifying exactly what the maps are. On the other hand, map encapsulation also plays a serious role; recognizing maps in itself and embodying them explicitly as nodes is one of OCP's important cognitive functions. The recognition of maps is done, in practice, by a combination of AI tools, including greedy datamining, and evolution/inference applied to time series pattern recognition within OCP.

Our treatment of maps here has been somewhat simplistic, but should be adequate to motivate the more pragmatic discussions of the topic in later chapters. Another, deeper way to formally describe maps is to observe that they are patterns in OCP, according to the formal theory of pattern described in The Hidden Pattern. This is needed in order to formalize the notion of predictive pattern mentioned above. Maps are not the only kind of pattern in OCP, but they are a particularly interesting kind, because of their relative simplicity and because of the relative ease with which the system can recognize and encapsulate them.

It must be emphasized that the major cognitive, dynamical patterns in a OCP system are not contained in the set of maps in the system. Rather, the dynamics *of* the set of maps in the system are the essence of the system's high-level cognitive dynamics. Maps are highly coherent, often narrowly focused repeated patterns; whereas most of thought is about transient patterns, or repeated patterns that are too fuzzy and abstract to be very *intense* if considered as maps (and not recognizable as maps by the system itself).

Chapter Four: Denoting Atoms (and Various Related Entities)

Meta-Language

This page confronts a generic notational issue pertaining to the OpenCog Prime docs generically. As always occurs when discussing (even partially) logic-based systems, there is some potential for confusion between logical relationships inside the system, and logical relationships being used to describe parts of the system. For instance, we can state as observers that two Atoms inside Novamente are equivalent, and this is different from stating that Novamente itself contains an Equivalence relation between these two Atoms. Our formal notation needs to reflect this difference.

Since we will not be doing any fancy mathematical analyses of Novamente structures or dynamics, there is no need to formally specify the logic being used for the Novamente metalanguage. Standard predicate logic may be assumed.

So, for example, we will say things like

```
(IntensionalInheritanceLink Ben monster).TruthValue.strength = .5
```

This is a metalanguage statement, which means that the strength field of the TruthValue object associated with the link (Instance Ben monster) is equal to .5. This is different than saying

```
EquivalenceLink
    ExOutLink
        GetStrength
        ExOutLink
            GetTruthValue
            IntensionalInheritanceLink Ben monster
    NumberNode 0.5
```

which refers to an equivalence relation represented inside Novamente. The former refers to an equals relationship observed by the author of the book, but perhaps never represented explicitly inside Novamente.

In the first example above we have used the C++ convention

```
structure_variable_name . field_name
```

for denoting elements of composite structures; this convention will be stated formally below.

In the second example we have used schema corresponding to TruthValue and Strength; these schema extract the appropriate fields from the Atoms they're applied to, so that e.g.

```
ExOutLink
    GetTruthValue
    A
```

returns the number

```
A . TruthValue
```

Following a convention from mathematical logic, we will also sometimes use the special symbol

```
|-
```

to mean implies in the metalanguage. For example, the first-order PLN deductive inference strength rule may be written

```
|}
Inheritance A B <sAB>
Inheritance B C <sBC>
```

```
|-
Inheritance A C <sAC>
```

where

```
sAC = sAB sBC + (1-sAB) ( sC - sB sBC ) / (1- sB )
```

This is different from saying

```
ForAll $A, $B, $C, $sAB, $sBC, $sAC

Implication_HOJ

    AND

        Inheritance $A $B <$sAB>

        Inheritance $B $C <$sBC>

    AND

        Inheritance $A $C <$sAC>

        $sAC = $sAB $sBC + (1-$sAB) ( $sC - $sB $sBC ) / (1- $sB )
```

which is the most natural representation of the independence-based PLN deduction rule (for strength-only truth values) as a logical statement within Novamente. In the latter expression the variables $A, $sAB, and so forth represent actual Variable Atoms within Novamente. In the former expression the variables represent concrete, non-Variable Atoms within Novamente, which however are being considered as variables within the metalanguage.

(Recall from the PLN book that a link labeled with "HOJ" refers to a "higher order judgment", meaning a relationship that interprets its relata as entities with particular truth values. For instance,

```
Implication
   Inh $X stupid <.9>
   Inh $X rich <.9>
```

means that if (Inh $X stupid) has a strength of .9, then (Inh $X rich) has a strength of .9)

Denoting Atoms

Atoms are the basic objects making up OpenCog knowledge. They come in various types, and are associated with various dynamics, which are embodied in MindAgents. Generally speaking Atoms are endowed with TruthValue and AttentionValue objects. They also sometimes have names, and other associated Values as previously discussed. In the following subsections we will explain how these are notated, and then discuss specific notations for Links and Nodes, the two types of Atoms in the system.

[edit] Names

In order to denote an Atom in discussion, we have to call it something. Relatedly but separately, Atoms may also have names within the OpenCog system. (As a matter of implementation, in the current OpenCog version, no Links have names; whereas, all Nodes have names, but some Nodes have a null name, which is conceptually the same as not having a name.)

(name,type) pairs must be considered as unique within each Unit within a OpenCog system, otherwise they can't be used effectively to reference Atoms. It's OK if two different OpenCog Units both have SchemaNodes named +, but not if one OpenCog Unit has two SchemaNodes both named + — this latter situation is disallowed on the software level, and is assumed in discussions not to occur.

Some Atoms have natural names. For instance, the SchemaNode corresponding to the elementary schema function + may quite naturally be named +. The NumberNode corresponding to the number .5 may naturally be named .5, and the CharacterNode corresponding to the character *c* may naturally be named c. These cases are the minority, however. For instance, a SpecificEntityNode representing a particular instance of + has no natural name, nor does a SpecificEntityNode representing a particular instance of c.

Names should not be confused with Handles. Atoms have Handles, which are unique identifiers (in practice, numbers) assigned to them by the OpenCog core system; and these Handles are how Atoms are referenced internally, within OpenCog, nearly all the time. Accessing of Atoms by name is a special case — not all Atoms have names, but all Atoms have Handles. An example of accessing an Atom by name is looking up the CharacterNode representing the letter "c" by its name "c". There would then be two possible representations for the word "cat":

1. this word might be associated with a ListLink — and the ListLink corresponding to "cat" would be a list of the Handles of the Atoms of the nodes named "c", "a", and "t".
2. for expedience, the word might be associated with a WordNode named "cat."

In the case where an Atom has multiple versions, each version has a VersionHandle, so that accessing an AtomVersion requires specifying an AtomHandle plus a VersionHandle. More on Handles will be presented in the chapter on the MindOS.

OpenCog never assigns Atoms names *on its own*; in fact, Atom names are assigned only in the two sorts of cases weve just mentioned

1. Via preprocessing of perceptual inputs (e.g. the names of NumberNode, CharacterNodes)
2. Via hard-wiring of names for SchemaNodes and PredicateNodes corresponding to built-in elementary schema (e.g. +, AND, Say)

If an Atom A has a name n in the system, we may write

```
A.name = n
```

On the other hand, if we want to assign an Atom an *external* name, we may make a meta-language assertion such as

```
L1 := (InheritanceLink Ben animal)
```

indicating that we decided to name that link L1 for our discussions, even though inside OpenCog it has no name.

In denoting nameless Atoms we may use arbitrary names like L1. This is more convenient than using a Handle based notation which Atoms would be referred to as [1], [3433322], etc.; but sometimes we will use the Handle notation as well.

Some ConceptNodes and conceptual PredicateNode or SchemaNodes may correspond with human-language words or phrases like *cat*, *bite*, and so forth. This will be the minority case; more such nodes will correspond to parts of human-language concepts or fuzzy collections of human-language concepts. In discussions in this book, however, we will often invoke the unusual case in which Atoms correspond to individual human-language concepts. This is because such examples are the easiest ones to discuss intuitively. The preponderance of named Atoms in the examples in the book implies no similar preponderance of named Atoms in the real OpenCog system. It is merely easier to talk about a hypothetical Atom named "cat" than it is about a hypothetical Atom (internally) named [434]. It is not impossible that a OpenCog system represents "cat" as a single ConceptNode, but it is just as likely that it will represent "cat" as a map composed of many different nodes without any of these having natural names. Each OpenCog works out for itself, implicitly, which concepts to represent as single Atoms and which in distributed fashion.

For another example,

```
ListLink
    CharacterNode "c"
    CharacterNode "a"
    CharacterNode "t"
```

corresponds to the character string

```
("c", "a", "t")
```

and would naturally be named using the string *cat*. In the system itself, however, this ListLink need not have any name.

Types

Atoms also have types. When it is necessary to explicitly indicate the type of an atom, we will use the keyword Type, as in

```
A.Type = InheritanceLink

N_345.Type = ConceptNode
```

On the other hand, there is also a built-in schema HasType which lets us say

```
EvaluationLink HasType A InheritanceLink
```

```
EvaluationLink HasType N_345 ConceptNode
```

This covers the case in which type evaluation occurs explicitly in the system, which is useful if the system is analyzing its own emergent structures and dynamics.

Truth Values

The truth value of an atom is a bundle of information describing how *true* the Atom is, in one of several different senses depending on the Atom type. It is encased in a TruthValue object associated with the Atom. Most of the time, we will denote the truth value of an atom in <>'s following the expression denoting the atom. This very handy notation may be used in several different ways.

A complication is that some Atoms may have CompositeTruthValues, which consist of different estimates of their truth value made by different sources, which for whatever reason have not been reconciled (maybe no process has gotten around to reconciling them, maybe they correspond to different truth values in different contexts and thus logically need to remain separate, maybe their reconciliation is being delayed pending accumulation of more evidence, etc.). In this case we can still assume that an Atom has a default truth value, which corresponds to the highest-confidence truth value that it has, in the Universal Context.

Most frequently, the notation is used with a single number in the brackets, e.g.

```
A <.4>
```

to indicate that the atom A has truth value .4; or

```
IntensionalInheritanceLink Ben monster <.5>
```

to indicate that the IntensionalInheritance relation between Ben and monster has truth value strength .5. In this case, <tv> indicates (roughly speaking) that the truth value of the atom in question involves a probability distribution with a mean of tv. The precise semantics of the strength values associated with OpenCog Atoms is described in Probabilistic Logic Networks. Please note, though: This notation does not imply that the only data retained in the system about the distribution is the single number .5.

If we want to refer to the truth value of an Atom in the context C, we can use the construct

```
ContextLink
  C
  A <_truth value_>
```

Sometimes, Atoms in OpenCog are labeled with two truth value components as defined by PLN: strength and weight-of-evidence. To denote these two components, we might write

```
IntensionalInheritanceLink Ben scary <.9,.1>
```

indicating that there is a relatively small amount of evidence in favor of the proposition that Ben is very scary.

If we want to denote a composite truth value (whose components correspond to different "versions" of the Atom), we can use a list notation, e.g.

```
IntensionalInheritanceLink Ben scary (<.9,.1>, <.5,.9> [h,123],<.6,.7> [c,655])
```

where e.g.

```
<.5,.9> [h,123]
```

denotes the TruthValue version of the Atom indexed by Handle [123]. The _h_ denotes that the AtomVersion indicated by the VersionHandle [h,123] is a Hypothetical Atom, in the sense described in the PLN book. Some versions may not have any index Handles.

The semantics of composite TruthValues are described in the PLN book, but roughly they are as follows. Any version not indexed by a VersionHandle is a "primary TruthValue" that gives the truth value of the Atom based on some body of evidence. A version indexed by a VersionHandle is either contextual or hypothetical, as indicated notationally by the c or h in its VersionHandle. So, for instance, if a TruthValue version for Atom A has VersionHandle [h,123] that means it denotes the truth value of Atom A under the hypothetical context represented by the Atom with handle [123]. If a TruthValue version for Atom A has VersionHandle [c,655] this means it denotes the truth value of Atom A in the context represented by the Atom with Handle [655].

Alternately, truth values may be expressed sometimes in <L,U,b> or <L,U,b,N> format, defined in terms of indefinite probability theory as defined in the PLN book. For instance,

```
IntensionalInheritanceLink Ben scary <.7,.9,.8,20>
```

has the semantics that *There is an estimated 80% chance that after 20 more observations have been made, the estimated strength of the link will be in the interval (.7,.9).*

```
Sometimes we will put the TruthValue indicator in a different place, e.g. using indent
notation,
IntensionalInheritanceLink <.5>
    Ben
    Monster
```

This is mostly useful when dealing with long and complicated constructions — which is however a fairly common case....

The notation may also be used to specify a TruthValue probability distribution, e.g.

```
A < g(.5,7,12)>
```

would indicate that the truth value of A is given by distribution g with parameters (5,7,12), or

```
A < M >
```

where M is a table of numbers would indicate that the truth value of A is approximated by the table M.

The <> notation for truth value is an unabashedly incomplete and ambiguous notation, but it is very convenient. If we want to specify, say, that the truth value strength of IntensionalInheritanceLink Ben monster is in fact the number .5, and no other truth value information is retained in the system, then we need to say

```
(IntensionalInheritanceLink Ben monster).TruthValue = [(strength, .5)]
```

(where a hashtable form is assumed for TruthValue objects, i.e. a list of name-value pairs). But this kind of issue will rarely arise and the <> notation will serve us very well.

We will also sometimes use the object-field style with TruthValues; for instance, if tv is a TruthValue object, tv.s means the strength field of tv.

Attention Values

The AttentionValue object associated with an Atom does not need to be notated nearly as often as truth value. When it does however we can use similar notational methods.

AttentionValues may have several components, but the two critical ones are called short-term importance (STI) and long-term importance (LTI). Furthermore, multiple STI values are retained: for each (Atom, MindAgent) pair there may be a Mind-Agent-specific STI value for that Atom. The pragmatic import of these values will become clear in a later chapter when we discuss attention allocation.

Roughly speaking, the long-term importance is used to control memory usage: when memory gets scarce, the atoms with the lowest LTI value are removed. On the other hand, the short-term importance is used to control processor time allocation: MindAgents, when they decide which Atoms to act on, will generally choose the ones that have proved most useful to them in the recent past, and additionally those that have been useful for other MindAgents in the recent past.

We will use the double bracket <<>> to denote attention value (in the rare cases where such denotation is necessary). So, for instance,

```
Cow_7 <<.5>>
```

will mean the node Cow_7 has an importance of .5; whereas,

```
Cow_7 <<STI=.1, LTI = .8>>
```

will mean the node Cow_7 has short-term importance = .1 and long-term importance = .8 .

Of course, we can also use the style

```
(IntensionalInheritanceLink Ben monster). AttentionValue = [(STI,.1), (LTI, 4)]
```

where appropriate.

Generally, specific denotations of AttentionValues come up much less often in discussion than specific denotations of TruthValues.

Links

Links are represented using a simple notation that has already occurred many times in this book. For instance,

```
Inheritance A B

Similarity A B
```

Note that here the symmetry or otherwise of the link is not implicit in the notation. SimilarityLinks are symmetrical, InheritanceLinks are not. When this distinction is necessary, it will be explicitly made.

Denoting Functions and Predicates

SchemaNodes and PredicateNodes contain functions internally; and Links may also be usefully be considered as functions. We now briefly discuss the notations we will use to denote functions in various contexts.

Firstly, we will make some use of the currying notation drawn from combinatory logic, in which adjacency indicates function application. So, for instance, using currying,

```
f x
```

means the function f evaluated at the argument x; and (f x y) means (f(x))(y). If we want to specify explicitly that a block of terminology is being specified using currying we will use the notation @[<expression>]; for instance

```
@[f x y z]
```

means

```
((f(x))(y))(z)
```

We will also frequently use conventional notation to refer to functions, such as f(x,y). Of course, this is consistent with the currying convention if (x,y) is interpreted as a list and f is then a function that acts on 2-element lists. We will have many other occasions than this to use list notation.

Also, we will sometimes use a non-curried notation, most commonly with Links, so that e.g.

```
InheritanceLink x y
```

does not mean a curried evaluation but rather means InheritanceLink(x,y). If we want to explicitly specify that an expression is non-curried, we will use the notation ![*expression*].

Execution Output Links

In the case where f refers to a schema, the occurrence of the combination f x in the system is represented by

```
ExOutLink f x
```

or graphically

```
      @
```

```
    /   \

  f       x
```

Note that, just as when we write

```
f (g x)
```

we mean to apply f to the result of applying g to x, similarly when we write

```
ExOutLink f (ExOutLink g x)
```

we mean the same thing.

So for instance

```
EvaluationLink (ExOutLink g x) y <.8>
```

means that the result of applying g to x is a predicate r, so that r(y) evaluates to True with strength .8.

This approach, in its purest incarnation, does not allow multi-argument schemata. Now, multi-argument schemata are never actually necessary, because one can use argument currying to simulate multiple arguments. However, this is often awkward, and things become simpler if one introduces an explicit tupling operator, which we call ListLink. Simply enough,

```
ListLink A1 … An
```

denotes an ordered list (A1 ,… ,An)

Execution Links

ExecutionLinks give the system an easy way to record acts of schema execution. These are ternary links of the form:

```
SchemaNode: S

Atom: A, B

ExecutionLink S A B
```

In words, this says the procedure represented by SchemaNode S has taken input A and produced output B.

There may also be schemata that do not take output, or do not take input. But these are treated as PredicateNodes, to be discussed below; their activity is recorded by EvaluationLinks, not ExecutionLinks.

The TruthValue of an ExecutionLink records how frequently the result encoded in the ExecutionLink occurs. Specifically,

- the TruthValue of (ExecutionLink S A B) tells you the probability of getting B as output, given that you have run schema S on input A
- the TruthValue of (ExecutionLink S A) tells you the probability that if S is run, it is run on input A

Often it is useful to record the time at which a given act of schema execution was carried out; in that case one uses the atTime schema, writing e.g.

```
atTime (ExecutionLink S A B) T
```

where T is a TimeNode, or else one uses an implicit method such as storing the time-stamp of the ExecutionLink in a core-level data-structure called the TimeServer. The implicit method is logically equivalent to explicitly using atTime, and is treated the same way by PLN inference, but provides significant advantages in terms of memory usage and lookup speed.

For purposes of logically reasoning about schema, it is useful to create binary links representing ExecutionLinks with some of their arguments fixed. We name these as follows:

```
ExecutionLink1 A B means: X so that ExecutionLink X A B

ExecutionLink2 A B means: X so that ExecutionLink A X B

ExecutionLink3 A B means: X so that ExecutionLink A B X
```

Finally, a SchemaNode may be associated with a structure called a Graph.

Where S is a SchemaNode,

```
Graph(S) = { (x,y): ExecutionLink S x y }
```

Sometimes, the graph of a SchemaNode may be explicitly embodied as a ConceptNode; other times, it may be constructed implicitly by a MindAgent in analyzing the SchemaNode (e.g. the inference MindAgent).

Note that the set of ExecutionLinks describing a SchemaNode may not define that SchemaNode exactly, because some of them may be derived by inference. This means that the model of a SchemaNode contained in its ExecutionLinks may not actually be a mathematical function, in the sense of assigning only one output to each input. One may have

```
ExecutionLink S X A <.5>

ExecutionLink S X B <.5>
```

meaning that the system does not know whether S(X) evaluates to A or to B. So the set of ExecutionLinks modeling a SchemaNode may constitute a non-function relation, even if the schema inside the SchemaNode is a relation.

Finally, what of the case where f x represents the action of a built-in system function f on an argument x? This is an awkward case in that it will not be necessary once the OCP system is revised so that all cognitive functions are carried out using SchemaNodes. However, in OCP 0.5 where most cognitive functions are carried out using

C++ MindAgent objects, if we want OCP to study its own cognitive behavior in a statistical way, we need BuiltInSchemaNodes that refer to MindAgents rather than to ComboTrees (or else, we need to represent MindAgents using ComboTrees, which will become practicable once we have a sufficiently efficient Combo interpreter). The semantics here is thus basically the same as where f refers to a schema. For instance we might have

```
ExecutionLink FirstOrderInferenceMindAgent (L1, L2) L3
```

where L1, L2 and L3 are links related by

```
L1
L2
   L3
```

according to the first-order PLN deduction rule.

Predicates

Predicates are related but not identical to schema, both conceptually and notationally. PredicateNodes involve *predicate schema* which output TruthValue objects. But there is a difference between a SchemaNode embodying a predicate schema and a PredicateNode, which is that a PredicateNode doesn't output a TruthValue, it adjusts its own TruthValue as a result of the output of its own internal predicate schema.

The record of the activity of a PredicateNode is given not by an ExecutionLink but rather by an:

```
EvaluationLink P A <t>
```

where P is a PredicateNode, A is its input, and <tv> is the truth value assumed by the EvaluationLink corresponding to the PredicateNode being fed the input A. There is also the variant

```
EvaluationLink P <tv>
```

for the case where the PredicateNode P embodies a schema that takes no inputs.

A simple example of a PredicateNode is the predicate GreaterThan. In this case we have, for instance

```
EvaluationLink GreaterThan 5 6 <0>

EvaluationLink GreaterThan 5 3 <1>
```

and we also have:

```
EquivalenceLink
    GreaterThan
    ExOutLink
        And
        ListLink
            ExOutLink
```

```
            Not
            LessThan
        ExOutLink
            Not
            EqualTo
```

We will also encounter many commonsense-semantics predicates such as isMale, with e.g.

```
EvaluationLink isMale Ben_Goertzel <1>
```

Schemata that return no outputs are treated as predicates, and handled using EvaluationLinks. The truth value of such a predicate, as a default, is considered as True if execution is successful, and False otherwise.

And, analogously to the Graph operator for SchemaNodes, we have for PredicateNodes the SatisfyingSet operator, defined so that the SatisfyingSet of a predicate is the set whose members are the elements that satisfy the predicate. Formally, that is:

```
S = SatisfyingSet P
```

means

```
TruthValue ( MemberLink X S )

equals

TruthValue ( EvaluationLink P X)
```

This operator allows the system to carry out advanced logical operations like higher-order inference and unification.

Denoting Schema and Predicate Variables

OCP sometimes uses variables to represent the expressions inside schemata and predicates, and sometimes uses variable-free, combinatory-logic-based representations. There are two sorts of variables in the system, either of which may exist either inside compound schema or predicates, or else in the Atomspace as VariableNodes:

It is important to distinguish between two sorts of variables that may exist in OCP:

- Variable Atoms, which may be quantified (bound to existential or universal quantifiers) or unquantified
- Variables that are used solely as function-arguments or local variables inside the "Combo tree" structures used inside some ProcedureNodes (PredicateNodes or SchemaNodes) (to be described below), but are not related to Variable Atoms

Examples of quantified variables represented by Variable Atoms are $X and $Y in:

```
ForAll $X <.0001>

    ExtensionalImplicationLink

        ExtensionalInheritanceLink $X human
```

```
        ThereExists $Y
            AND
                ExtensionalInheritanceLink $Y human
                EvaluationLink parent_of ($X, $Y)
```

An example of an unquantified Variable Atom is $X in

```
ExtensionalImplicationLink <.3>
    ExtensionalInheritanceLink $X human
    ThereExists $Y
        AND
            ExtensionalInheritanceLink $Y human
            EvaluationLink parent_of ($X, $Y)
```

This ImplicationLink says that 30% of humans are parents: a more useful statement than the ForAll Link given above, which says that it is very very unlikely to be true that all humans are parents.

We may also say, for instance,

```
SatisfyingSet( EvaluationLink eats (cat, $X)
```

to refer to the set of X so that eats(cat, X).

On the other hand, suppose we have the implication

```
Implication
    Evaluation f $X
    Evaluation
        f
        ExOut reverse $X
```

where f is a PredicateNode embodying a mathematical operator acting on pairs of NumberNodes, and reverse is an operator that reverses a list. So, this implication says that the f predicate is commutative. Now, suppose that f is grounded by the formula

```
f(a,b) = (a > b – 1)
```

embodied in a Combo Tree object, stored in the ProcedureRepository and linked to the PredicateNode for f. These f-internal variables, which I have written here using the letters a and b, are not VariableNodes in the OCP

AtomTable. The notation we use for these within the textual Combo language, that goes with the Combo Tree formalism, is to replace a and b in this example with #1 and #2, so the above grounding would be denoted

```
f –> (#1 > #2 – 1)
```

Input and Output Type Restrictions

Each ProcedureNode (SchemaNode or PredicateNode) has certain input type restrictions, and certain output type restrictions. In other words, a schema doesn't usually act on a general Atom, but will often only meaningfully act on a particular type of Atom. And a schema doesn't usually output a general Atom, but will often only output a particular type of Atom it must come with.

This notion of type restriction may be formalized by considering each Procedure/InstanceNode to be associated with:

- An input type restriction ITR
- An output type restriction OTR

These type restrictions ITR and OTR are themselves predicates. It is unnecessary for these to be Atoms (in fact this would lead to an infinite regress), but there should be special schemata

```
GetInputTypeRestriction()

GetOutputTypeRestriction()
```

that produce schema embodying the ITR and OTR (respectively) of their argument SchemaNodes. The ITR and OTR can be expanded into PredicateNodes in cases where this is appropriate.

When necessary, type restrictions may be represented as relationships of the form

```
Type T
```

where T is a ConceptNode representing an Atom type. Here, Type is a predicate so that Type T is itself a PredicateNode, and the semantics is

```
EvaluationLink ((Type T) x) iff x is of type T
```

There may also be more subtle type restrictions. For instance, what if the input type of X is the set of SchemaNodes with input type Y and output type Z? Then

```
ITR(X) W
equals
AND
    EquivalenceLink (ITR(W)) Y
    EquivalenceLink (OTR(W)) Z
```

A ProcedureNode X can only be involved in a relationship

```
ExecutionOutputLink X Y
```

with an Procedure/InstanceNode Y if

```
OTR(X) W = True implies ITR(Y) W = True
```

or, put more simply, if

```
OTR(X) implies ITR(Y)
```

A non-ProcedureNode Z can only have an ExOutLink pointing to a ProcedureNode X if

```
ITR(X) Z equals True
```

These conditions must be checked whenever ExOutLinks are created.

In the current OCP version, it is assumed that type restrictions are always crisp, not probabilistically truth-valued. This assumption may be revisited in a later version of the system.

Links as Predicates

It is conceptually important to recognize that OCP link types may be interpreted as predicates. For instance, when one says

```
InheritanceLink cat animal <.8>
```

indicating an Inheritance relation between cat and animal with a strength .8, effectively one is declaring that one has a predicate giving an output of .8. Depending on the interpretation of InheritanceLink as a predicate, one has either the predicate

```
InheritanceLink cat $X
```

acting on the input

```
animal
```

or the predicate

```
InheritanceLink #1 animal
```

acting on the input

```
cat
```

or the predicate

```
InheritanceLink #1 #2
```

acting on the list input

```
(cat, animal)
```

This means that, if we wanted to, we could do away with all Link types except OrderedLink and UnorderedLink, and represent all other Link types as PredicateNodes embodying appropriate predicate schema.

This is not the approach taken in the current codebase. However, the situation is somewhat similar to that with CIM-Dynamics:

- In future we will create a revision of OCP that regularly revises its own vocabulary of Link types, in which case an explicit representation of link types as predicate schema will be appropriate.
- In the shorter term, it can be useful to treat link types as *virtual predicates,* meaning that one lets the system create SchemaNodes corresponding to them, and hence do some *meta level reasoning* about its own link types.

Representation of Cognitive Processes as Schemata

OCP's CIM-Dynamics provide it with an initial set of cognitive tools, with which it can learn how to interact in the world. One of the jobs of this initial set of cognitive tools, however, is to create better cognitive tools. Initially, this takes the form of *cognitive schema learning* — the learning of schemata carrying out cognitive processes in more specialized, context-dependent ways than the general CIM-Dynamics do. Eventually, these cognitive schema will rival the CIM-Dynamics in complexity, and will replace the CIM-Dynamics altogether, leaving the system to operate almost entirely based on cognitive schemata. This long-term vision of OCP as a self-modifying, self-programming system will be revisited later on.

In order to make the process of cognitive schema learning easier, we provide a number of elementary schemata embodying the basic cognitive processes contained in the CIM-Dynamics. Of course, cognitive schemata need not use these — they may embody entirely different cognitive processes than the CIM-Dynamics. Eventually, we want the system to discover better ways of doing things than anything even hinted at by its initial CIM-Dynamics. But for the initial phases or the system's schema learning, it will have a much easier time learning to use the basic cognitive operations as the initial CIM-Dynamics, rather than inventing new ways of thinking from scratch!

For instance, we provide a number of elementary schemata corresponding to inference operations, such as

```
Schema: Deduction
    Input InheritanceLink: X, Y
    Output InheritanceLink
```

The inference CIM-Dynamics apply this rule in certain ways, designed to be reasonably effective in a variety of situations. But there are certainly other ways of using the deduction rule, outside of the basic control strategies embodied in the inference CIM-Dynamics. By learning schemata involving the Deduction schema, the system can learn special, context-specific rules for combining deduction with concept-formation, association-formation and other cognitive processes. And as it gets smarter, it can then take these schemata involving the Deduction schema, and replace it with a new schema that eg. contains a context-appropriate deduction formula.

Eventually, to support cognitive schema learning, we will want to cast the hard-wired CIM-Dynamics as cognitive schemata, so the system can see what is going on inside them. Pragmatically, what this requires is coding versions of the CIM-Dynamics in Sasha rather than C++, to facilitate the studying of the system, even if

the C++ versions are the ones actually executed until the system's self-modificatory learning has proceeded so far that it has rendered its original CIM-Dynamics useless.

But even prior to this kind of fully *cognitively transparent* implementation, the system can still reason about its use of different mind dynamics by considering each CIM-Dynamic as a *virtual Procedure* with a real SchemaNode attached to it. This can lead to some valuable learning, with the obvious limitation that in this approach the system is thinking about its CIM-Dynamics as *black boxes* rather than being equipped with full knowledge of their internals.

Eliminating Variables Using Combinators

One of the most important aspects of combinatory logic, from a OCP perspective, is that it allows one to represent arbitrarily complex procedures and patterns without using variables in any direct sense. In OCP, variables are optional, and the choice of whether or how to use them may be made (by OCP itself) on a contextual basis.

This page deals with the representation of *variable expressions* in a variable-free way, in a OCP context. The general theory underlying this is well-known, and is usually expressed in terms of the elimination of variables from lambda calculus expressions (*lambda lifting*). Here we will not present this theory but will restrict ourselves to presenting a simple, hopefully illustrative example, and then discussing some conceptual implications.

Why Eliminating Variables is So Useful

Before launching into the specifics, a few words about the general utility of variable-free expression may be worthwhile.

Some expressions look simpler to the trained human eye with variables, and some look simpler without them. However, the main reason why eliminating all variables from an expression is sometimes very useful, is that there are automated program-manipulation techniques that work much more nicely on programs (schemata, in OCP lingo) without any variables in them.

As will be discussed later (e.g. in the chapter on evolutionary learning, although the same process is also useful for supporting probabilistic reasoning on procedures), iIn order to mine patterns among multiple schema that all try to do the same (or related) things, we want to put schema into a kind of "hierarchical normal form." The normal form we wish to use generalizes Holman's Elegant Normal Form (which is discussed in Moshe Looks' PhD thesis) to program trees rather than just Boolean trees.

But, putting computer programs into a useful, nicely-hierarchically-structured normal form is a hard problem — it requires one have a pretty nice and comprehensive set of *program transformations*.

But the only general, robust, systematic program transformation methods that exist in the computer science literature require one to remove the variables from one's programs, so that one can use the theory of functional programming (which ties in with the theory of monads in category theory, and a lot of beautiful related math).

So: In large part, we want to remove variables so we can use functional programming tools normalize programs into a standard and pretty hierarchical form, so we can mine patterns among them effectively.

However, we don't *always* want to be rid of variables, because sometimes, from a logical reasoning perspective, theorem-proving is easier with the variables in there. (Sometimes not.)

So, we want to have the option to use variables, or not.

An Example of Variable Elimination

Consider the PredicateNode

```
AND
  InheritanceLink X cat
  eats X mice
```

Here we have used a *syntactically sugared* representation involving the variable X. How can we get rid of the X?

Recall the C combinator, defined by

```
C f x y = f y x
```

Using this tool,

```
InheritanceLink X cat
```

becomes

```
C InheritanceLink cat X
```

and

```
(eats X) mice
```

becomes

```
C (eats) mice X
```

so that overall we have

```
AND
  C InheritanceLink cat
  C eats mice
```

where the C combinators essentially give instructions as to where the *virtual argument* X should go.

In this case the variable-free representation is basically just as simple as the variable-based representation, so there is nothing to lose and a lot to gain by getting rid of the variables. This won't always be the case — sometimes execution efficiency will be significantly enhanced by use of variables.

Inheritance Between Higher-Order Types

This page deals with the somewhat subtle matter of Inheritance between higher-order types. This is needed, for example, when one wants to cross over or mutate two complex schemata, in an evolutionary learning context. One encounters questions like: When mutation replaces a schema that takes integer input, can it replace it with one that takes general numerical input? How about vice versa? These questions get more complex when the inputs and outputs of schema may themselves be schema with complex higher-order types. However, they can be dealt with elegantly using some basic mathematical rules.

(NOTE: this material was largely worked out by Lukasz Kaiser and Cassio Pennachin, in collaboration with Ben Goertzel)

Denote the type of a mapping from type T to type S, as T —> S. Use the shorthand *inh* to mean *inherits from.* Then the basic rule we use is that

```
T1 -> S1 inh T2 -> S2

iff

T2 inh T1
S1 inh S2
```

In other words, we assume higher-order type inheritance is countervariant. The reason is that, if R1 = T1 —> S1 is to be a special case of R2 = T2 —> S2, then one has to be able to use the latter everywhere one uses the former. This means that any input R2 takes, has to also be taken by R1 (hence T2 inherits from T1). And it means that the outputs R2 gives must be able to be accepted by any function that accepts outputs of R1 (hence S1 inherits from S2).

This type of issue comes up in programming language design fairly frequently, and there are a number of research papers debating the pros and cons of countervariance versus covariance for complex type inheritance. However, for the purpose of schema type inheritance in OCP, the greater logical consistency of the countervariance approach holds sway.

For instance, in this approach, INT —> INT is not a subtype of NO —> INT (where NO denotes FLOAT), because NO —> INT is the type that includes all functions which take a real and return an int, and an INT —> INT does not take a real. Rather, the containment is the other way around: every NO —> INT function is an example of an int ? int function. For example, consider the NO ? INT that takes every real number and rounds it up to the nearest integer. Considered as an INT —> INT function, this is simply the identity function: it is the function that takes an integer and rounds it up to the nearest integer.

Of course, tupling of types is different, it's covariant. If one has an ordered pair whose elements are of different types, say (T1, T2), then we have

```
(T1 , S1) inh (T2, S2)

 iff

T1 inh T2

S1 inh S2
```

As a mnemonic formula, we may say

```
(general –> specific ) inherits from (specific –> general)

(specific, specific) inherits from (general, general)
```

In schema learning, we will also have use for abstract type constructions, such as

```
(T1, T2) where T1 inherits from T2
```

Notationally, we will refer to variable types as Xv1, Xv2, etc., and then denote the inheritance relationships by using numerical indices, e.g. using

```
[1 inh 2]
```

to denote that

```
Xv1 inh Xv2
```

So for example,

```
( INT, VOID ) inh ( Xv1 , Xv2 )
```

is true, because there are no restrictions on the variable types, and we can just assign Xv1 = INT, Xv2 =VOID.

On the other hand,

```
( INT, VOID ) inh ( Xv1, Xv2 ), [ 1 inh 2 ]
```

is false because the restriction Xv1 inh Xv2 is imposed, but it's not true that INT inh VOID.

The following list gives some examples of type inheritance, using the elementary types INT, FLOAT (FL), NUMBER (NO), CHAR and STRING (STR), with the elementary type inheritance relationships

- INT inh NUMBER
- FLOAT inh NUMBER
- CHAR inh STRING
- (NO —> FL) inh (INT —> FL)
- (FL —> INT) inh (FL —> NO)
- ((INT —> FL) —> (FL —> INT)) inh ((NO —> FL) —> (FL —> NO))

Advanced Schema Manipulation

On this page we describe some special schema for manipulating schema, which seem to be very useful in certain contexts.

Listification

First, there are two ways to represent n-ary relations in OCP's Atom level knowledge representation language: using lists as in

```
f_list (x1, …, xn )
```

or using currying as in

```
f_curry x1, …, xn
```

To make conversion between list and curried forms easier, we have chosen to introduce special schema (combinators) just for this purpose:

```
Listify f = f_list so that f_list (x1, …, xn ) = f x1 … xn

Unlistify Listify f = f
```

For instance

```
kick_curry Ben Ken
```

denotes

```
(kick_curry Ben) Ken
```

which means that kick is applied to the argument Ben to yield a predicate schema applied to Ken. This is the curried style. The list style is

```
kick_List (Ben, Ken)
```

where kick is viewed as taking as an argument the List (Ben, Ken). The conversion between the two is done by

```
listify kick_curry = kick_list

unlistify kick_list = kick_curry
```

As a more detailed example of unlistification, let us utilize a simple mathematical example, the function

```
(X- 1 )^2
```

As noted there, if we use the notations -, *, *Square* and *Power* to denote SchemaNodes embodying the corresponding operations, then then this formula may be written in variable-free node-and-link form as

```
ExOutLink
    pow
    ListLink
        ExOutLink
            -
            ListLink
                X
                1
        2
```

But to get rid of the nasty variable X, we need to first unlistify the functions pow and -, and then apply the C combinator a couple times to move the variable X to the front. This is accomplished as follows:

```
pow [ - [x,1], 2 ]
unlistify pow (-[x,1]) 2
C (unlistify pow) 2 (-[x,1])
C (unlistify pow) 2 ((unlistify -) x 1)
C (unlistify pow) 2 (C (unlistify -) 1 x)
B ( C (unlistify pow) 2) ( C (unlistify — ) 1) x
```

yielding the final schema

```
B ( C (unlistify pow) 2) ( C (unlistify — ) 1)
```

By the way, a variable-free representation of this schema in OCP would look like

```
ExOutLink
    B
    ExOutLink
        ExOutLink
            ExOutLink
                C
                ExOutLink
                    unlistify
                    pow
            2
            ExOutLink
                C
                ExOutLink
                    unlistify
                    -
        1
```

This representation may be difficult for the reader who lacks experience with functional programming. It involves the C combinator (discussed above), and the unlistify operator, defined so that e.g.

```
((unlistify f) x) (y) = f (x, y)
```

These concepts will be discussed in detail later, and used to do this example (and others) with more thorough explanation.

The main thing to be observed, however, is that the introduction of these extra schema lets us remove the variable X. The size of the schema is increased slightly in this case, but only slightly — an increase that is well-justified by the elimination of the many difficulties that explicit variables would bring to the system. Furthermore, there is a shorter rendition which looks like

```
ExOutLink
    B
    ExOutLink
        ExOutLink
            ExOutLink
                C
                pow_curried
            2
            ExOutLink
                C
                -_curried
        1
```

This rendition uses alternate variants of - and pow schema, labeled -_curried and pow_curried, which do not act on lists but are *curried* in the manner of combinatory logic and Haskell. It is 11 lines whereas the variable-bearing version is 8 lines, a trivial increase in length that brings a lot of operational simplification.

Finally, in case the indent notation is confusing, we give a similar example in a more standard predicate logic notation. We will translate the variable-laden predicate

```
likes(y,x) AND likes(x,y)
```

into the equivalent combinatory logic tree. Assume that the two inputs are going to be given to us as a list. Now, the combinatory logic representation of this is

```
S (B AND (B likes (listify C)) likes
```

using the standard convention of left-associativity for @.

We now show how this would be evaluated to produce the correct expression:

```
S (B AND (B likes (listify C)) likes (x,y)
```

S gets evaluated first, to produce

```
B AND (B likes (listify C)) (x,y) (likes (x,y))
```

now the first B

```
AND (B likes (listify C) (x,y)) (likes (x,y))
```

now the second one

```
AND (likes ((listify C) (x,y)) (likes (x,y)))
```

now the listified C

```
AND (likes (y,x)) (likes (x,y)) )
```

which is what we wanted.

Argument Permutation

In dealing with List relationships, there will sometimes be use for an argument-permutation operator

```
ListOfAtom PermuteArgumentList(Permutation p, ListOfAtom A)
```

The semantics are that

```
(P f) (v1,…, vn ) = f ( P (v1,…, vn ) )
```

where P is a permutation on n letters. This deals with the case where we want to say, for instance that

```
Equivalence parent(x,y) child(y,x)
```

Instead of positing variable names x and y that span the two relations parent(x,y) and child(y,x), what we can instead say in this example is

```
Equivalence parent ( PermuteArgumentList {2,1} child )
```

For the case of two-argument functions, argument permutation is basically doing on the list level what the C combinator does in the curried function domain. On the other hand, in the case of n-argument functions with n>2, argument permutation doesn't correspond to any of the standard combinators.

Chapter Five: The Combo Language

The Combo Language

This page and the two that it supervenes over

- OpenCogPrime:ComboAndOCP
- OpenCogPrime:ComboAndInference
- OpenCogPrime:ComboAndLocalVariables

give a high-level conceptual discussion of the Combo language, which is a simple programming language created for representing the procedure objects that ground OpenCogPrime:GroundedProcedureNodes.

The Combo language is constantly evolving and the reference point for the details is the OpenCog : Combo page, which is supposed to be kept in coordination with the code. This page and its children are intended to focus more on the overall ideas underlying Combo.

The name "Combo" originated because the first version of Combo made heavy use of combinatory logic ideas. Subsequent versions have drifted a bit from this origin but the name has been retained because it sounds sort of wizzy, it's still relevant (as combinatory logic ideas are still present), and no clearly better name has come up.

Finally an authorial/historical note: Unlike most pages of this OpenCogPrime wiki, the initial version of this one was not mainly written by Ben Goertzel. Rather, the first draft was assembled by Ben Goertzel from prior documents written by Moshe Looks, the primary creator of the Combo language.

The "contents" of most OCP nodes is given by the relationships in which they're involved.

Basic perceptual node types are one exception to this: the contents of an NumberNode is a number, the contents of a CharacterNode is a character, etc. The other exceptions are the ProcedureNode types, SchemaNode and PredicateNode. In particular, these node types have subtypes GroundedSchemaNode and GroundedPredicateNode. These nodes correspond to Procedure objects, which wrap up objects called Combo trees. (Implementation-wise, in OCP the Procedure objects are stored in a special structure called the ProcedureRepository, in which they are indexed by the Handle of the Atom that corresponds to them.) These Combo trees are the objects that OCP uses to directly guide the actions that it learns how to do (both internally and externally). When inference or evolution processes figure out how to carry out some action, the knowledge achieved is stored not only in the form of declarative knowledge suitable for ongoing reasoning, but also in the form of procedural knowledge suitable for actual execution (i.e. Combo trees). The purpose of this page and its children is to give rough indications of the answers to a few basic, obvious questions about Combo trees: how they are constructed, how they are interpreted, and how they relate with the rest of OCP.

Structurally, Combo trees are data structures that are trees or dags (directed acyclic graphs) whose internal nodes are either

- @ symbols (denoting "function application," as defined in the earlier discussion of combinatory logic)
- function labels (indicating that the function denoted should be applied to the arguments denoted by the child nodes), or
- LIST operators (indicating that the child nodes should be arranged in a list)

and whose leaf nodes are either

- Elementary data items (e.g. numbers, characters or strings), or
- Functions, known as "elementary schemata" (elementary procedures)
- Input variables

The collection of elementary schemata used is known as the "elementary schema vocabulary." The precise optimal schema vocabulary for OCP is only going to be determined based on experimentation, and may vary based on the given OCP system’s application domain. In this chapter we will run through a significant portion of our working set of elementary schema functions, explaining the use of each one and giving examples.

Textual Representations of Combo

Raw Combo

Combo trees may be represented directly as strings using a set of typographical conventions, and this gives rise to a simple programming language, which we call Combo, or else (if there is a need to distinguish it from other languages built on top of Combo) "raw Combo."

Raw Combo is a bare-bones programming language created with the following primary design constraints: to allow complex programs to be represented concisely, and to be amenable to evolutionary programming and probabilistic reasoning methods. These constraints have led to a unique style of programming, in which the fundamental unit of programming is the function (a lamba-abstraction), but within function definitions, combinators (abstract tree rewriting rules) are frequently used, rather than lambda abstractions.

It is actually possible to write programs in the raw Combo language, although we don't recommend it except for very small and simple cases. Raw Combo has the advantage of mapping directly into OCP structures, but it has many disadvantages from a usability-by-humans perspective, for example the lack of local variables.

The lack of usability of raw Combo has led to a couple efforts to append more syntactic sugar onto the Combo framework.

The Sasha Language

In a 2005 collaboration, Lukasz Kaiser, Moshe Looks, Ben Goertzel and others created a language called Sasha, which has a much higher level of usability than raw Combo and compiles into the latter. As compared to raw Combo, Sasha can more easily be used to write nontrivially complex OCP or NCE schemata and predicates. However, development on Sasha was frozen in early 2005 due to lack of funding, and in practice when coding schema by hand needed to be done for the NCE, we have used raw Combo rather than Sasha, so far.

There are pages on the OCP wiki site informally describing Sasha

- ComboSashaShorthands
- GeneralSashaDescription
- SashaExamples

but unless Sasha development is resurrected these will be of purely historical interest.

Combo LISP

In 2007-2008, some work was done by Moshe Looks toward the end of creating a new LISP interpreter/compiler that outputs Combo trees. However, this project was suspended in Spring 2008 in favor of an alternate approach of taking an existing LISP interpreter and modifying it to output Combo trees. This work is currently in progress.

Making Combo Richer

Currently Combo trees are fairly limited in their representational mechanisms. Of course, they can represent anything (combinatory logic being a Turing-complete formalism), but the challenge is to craft a language in which cognitively simple procedures have simple (compact, easily vary-able and manipulable) representations.

Turner Combinators

One potential way to make Combo richer is simply to add powerful combinators to the elementary schema vocabulary. For instance, the so-called "Turner combinators" have a lot of power and we have not yet experimented with them in the OCP context.

Local Variables

One of the shortcomings of the Combo versions implemented so far has to do with local variables (a separate, though related, issue to input variables ... note that a fully pure combinatory logic based language would have only one input for each program, whereas we have never made a Combo version that pure, always allowing the possibility of multiple input variables). While in many cases it is preferable from a standpoint of automatic program learning and inference to avoid use of local variables, in some cases it provides a dramatic increase in simplicity, such that avoiding them begins to seem more dogmatic than elegant. But on the other hand, if one is going to incorporate local variables in Combo, one needs to decide how to do it, and there are many possibilities.

The approaches we've discussed in most detail are described on the page LocalVariablesInCombo.

Making Combo Programs Efficient

At the moment (July 2008) the Combo interpreter used in the OCP codebase is not tremendously efficient, which is an issue that needs to be resolved eventually. For instance, in the long term we would like to rewrite the OCP MindAgents in (some variant of, or something that compiles into) Combo, but that would not be pointful now because a MindAgent written in Combo would run too slowly to be practicable. This sort of issue will get practical attention within the OCP project once the cognition code is mature enough to adapt, improve and learn from the Combo versions of MindAgents.

Supercompiling Combo Programs

An interesting approach to solving this problem is to "supercompile" Combo into efficient C++ code for rapid execution. Some preliminary experiments of this nature were done in 2004, supercompiling simple Combo programs into efficient Java using the Java Supercompiler created by Supercompilers LLC (see the whitepapers at http://www.supercompilers.com for an overview of supercompilation technology). These experiments are outside the scope of the OCPWikiBook per se, but are informally described on some other pages on the OCP wiki site:

- SashaAndSupercompilation
- SupercompilingSashaOutput
- SupercompilingSchemaToSasha

That work, done in 2004, was rather promising and will be written up for publication when someone finds the time and motivation. Unfortunately, due to funding issues it was never really completed.

Combo and OpenCog Prime

Now we discuss the use of Combo to access and manipulate OCP structures.

Combo takes a fairly simple view of OCP's knowledge representation. The basic OCP units are Atoms. Every atom has a truth value, and an atom type from a fixed hierarchy of types. The top level of the type heirarchy divides all atoms into nodes or links, with many node and link subtypes (e.g., concept node, inheritance link, etc...). All nodes have a name (string), all links have n-ary outgoing sets of other atoms. Every atom lives in an atom table, a data structure which can be queried to access its atoms. Note that links in one atom table can have atoms from another table in their target sets.

Grounded procedure nodes have either Combo trees or C++ code associated with them (see GroundedProcedures). The Combo tree associated with a ProcedureNode may carry out basic logical or arithmetic operations, or it may access and manipulate OCP structures using the methods described herein.

OCP truth values have multiple subtypes (e.g., distributional truth values). At a minimum, they contain strength and confidence parameters (two reals in [0,1]). Count, a natural number, is sometimes used, which is related monotonically to confidence.

TODO: Incorporate Atom versions into Combo. -- Ben G, Jan 7 2007 (still not done, July 2008)

OCP Atoms, Truth Values, Atom Types, and Atom Tables in Combo and Sasha

The section describes the interface between Combo and the OCP AtomTable, in a general way. First of all, we have the following types (within Combo):

```
OCAtom, OCNode, OCLink, OCTruthValue, OCType, OCAtomTable
```

OCNode and OCLink are subtypes of OCAtom; every OCAtom is either an OCNode or an OCLink. All of the functions decribed below throw exceptions if their type requirements are not met.

The default OCAtomTable is accessible with:

```
AtomTable
```

OCTypes can be constructed from strings with:

```
OCType atom_type [type_name : string]
```

(throws an exception if the type isn't known to the core)

The naming convention for atom types is no spaces with the first letter of each word capitalized (JavaStyleClassNames).

Type inheritance can be checked with:

```
bool inherits_type [first_type : OCType] [second_type : OCType]
```

OCTruthValues currently come in only one form, SimpleTV, and are constructed with:

```
OCTruthValue simple_tv [strength : float] [confidence : float]
```

(note confidence not count!)

OCTruthValue properties can be accessed with:

```
float strength [tv : OCTruthValue]
float confidence [tv : OCTruthValue]
nat count [tv : OCTruthValue]
```

Atoms can be accessed with:

```
OCNode get_node [type : OCType] [name : string]
OCLink get_link [type : OCType] [outgoing : list <OCAtom>]
```

If the atom doesn't exist, it will be created with a default truth value, which is guaranteed to have zero confidence (and therefore zero count). To query an atom's existance, you can use:

```
bool exists_node [type : OCType] [name : string]
bool exists_link [type : OCType] [outgoing : list <OCAtom>]
```

Atoms can be created with:

```
OCNode create_node [type : OCType] [name : string] [tv : OCTruthValue]
OCLink create_link [type : OCType] [outgoing : list<OCAtom>] [tv : OCTruthValue]
```

If the atom already exists, it will be replaced. The create_node command produces an error if you try to use it to create a node with a grounded procedure type or built-in type (GroundedPredicateNode, GroundedSchemaNode, BuiltInConceptNode and all subtypes therof). For creating grounded combinator tree procedures, see Grounded Procedures.

Note that all of these functions operate on the default atom table (AtomTable). There are corresponing functions X_on, which take an atom table as their first argument, e.g.:

```
OCNode create_node_on [table : OCAtomTable] [type : OCType] [name : string] [tv :
OCTruthValue]
```

Atom properties can be accessed with:

```
OCAtomTable get_table [atom : OCAtom]
OCType get_type [atom : OCAtom]
OCTruthValue get_tv [atom : OCAtom]
string get_name [node : OCNode]
```

```
list<OCAtom> get_outgoing [link : OCLink]
```

Note that get_name can only be called on nodes, and get_outgoing can only be called on links.

Atom properties can be set with:

```
OCAtom set_tv [atom : OCAtom] [tv : OCTruthValue]
OCAtom set_table [atom : OCAtom] [table : OCAtomTable]
```

which return the atom they are called with.

As a shorthand for accessing variable nodes, `$varname` can be used, which is equivalent to:

```
get_node (atom_type "VariableNode") "varname"
```

The Combo language (see GeneralComboDescription) makes it easy to define shorthands, which are used to illustrate the more complex commands described below.

In output from the Combo interpreter, OCNodes are denoted using the syntax:

```
#name:type
```

and OCLinks with the syntax

```
#type(arg1,arg2,...,argN)
```

For conciseness, this syntax is used in some of the following exposition

Built-in Nodes

There are some OCP nodes which are created automatically before combo has started.

The following are built-in predicates and schema (see Grounded Procedures for an explanation):

```
#AND:BuiltInSchemaNode
#OR:BuiltInSchemaNode
#NOT:BuiltInSchemaNode
#ForAll:BuiltInPredicateNode
#ThereExists:BuiltInPredicateNode
```

There are also built-in concept nodes corresponding to all atom types, representing the set of all of the atoms of that type. This allows reflective expressions to be written. These are named like:

```
#InheritanceLink:BuiltInConceptNode
#SchemaNode:BuiltInConceptNode
#ConceptNode:BuiltInConceptNode
```

etc...

Note that "set of all the atoms of that type" includes atoms of subtypes as well. So for example, the BuiltInConceptNode named "ConceptNode" is a member of the set that it represents.

Procedures

The ProcedureNode hierarchy is as follows:

```
ProcedureNode
+PredicateNode
++GroundedPredicateNode
+++BuiltInPredicateNode
+++ComboPredicateNode
+SchemaNode
++GroundedSchemaNode
+++BuiltInSchemaNode
+++ComboSchemaNode
```

That is, BuiltInSchemaNode is a subtype of GroundedSchemaNode, which is a subtype of SchemaNode, etc...

Predicates are functions that may take in arguments, and produce a truth value as output. The semantics are that a predicate node is linked to the its list of argument(s) with an evaluation link, the truth value of which is output of the predicate on those arguments. So for example, we can say, using the shorthands given above:

```
create_link (atom_type "EvaluationLink")
                                (cons (pred "eats") (cons (list (cons (concept "cat") (cons
(concept "fish") nil))) nil))
                                (simple_tv .99 .99)
```

i.e., `eats(cat,fish).`

(Note that, as a syntactic simplification, in Combo we allow e.g.

```
EvaluationLink eats cat mouse
```

as a shorthand for

```
EvaluationLink eats (cat, mouse)
```

since there is no possibility for ambiguity here.)

Schemata are more general than predicates, and can produce any kind of output. Schema nodes are linked to their arguments *and* outputs via execution links. So for example:

```
create_link (atom_type "ExecutionLink")
                                (cons (schema "meaning_of") (cons (list (cons (concept
"life") nil)) nil))
                                (simple_tv 1 1)
```

i.e., `meaning_of(life)=42`. Note that even though the meaning_of schema is only taking a single argument (life), the argument is still given inside a list link rather than directly, for consistancy.

1. GroundedProcedures

Grounded Procedures

Grounded procedures are those that are associated with procedural knowledge that allows their outputs to be computed with certainty. Built in procedures have associated C++ code and cannot be modified by Combo. Combo procedures (ComboPredicateNodes and ComboSchemaNodes) have associated combo trees.

Combinator Tree Procedures

Combinator tree procedures can be created with:

```
OCNode create_predicate [name : string] [tree : combo] [tv : OCTruthValue]
OCNode create_schema [name : string] [tree : combo] [tv : OCTruthValue]
```

(use create_predicate_on and create_schema_on to specify an atom table).

Note that the tree argument can be any valid Combo expression. For example:

```
fuzzy_or(2):=simple_tv (max (strength (get_tv #1)) (strength (get_tv #2)))
                                                    (max (confidence (get_tv #1))
(confidence (get_tv #2)))
create_predicate "fuzzyOr" fuzzy_or (simple_tv .66 1)
create_schema "combinatorMashUp" (S B (S S K (I I B S S))) (simple_tv .5 .5)
```

Creation of a grounded combinator tree procedure is similar to the definition of a function. The differences are that procedure creation can occur within a larger tree, procedures are stored in the OCP core, and procedures don't have arities stored with them. Note that, like function definition, the combinator tree argument of a procedure is only evaluated when the procedure is invoked (see below for commands dealing with procedure invocation). This means that, for example:

```
create_predicate "randomTV" (simple_tv random random) (simple_tv 0.5 1)
```

stores a predicate that generates a random tv each time. To compute a single random TV and store it, you can say:

```
randomTV:=(simple_tv random random)
create_predicate "randomTV" randomTV (simple_tv 0.5 1)
```

since evaluation of constants (:=) is eager.

The combinator tree associated with a combinator tree procedure can be accessed via:

```
combo get_predicate [grounded_predicate : OCNode]
combo get_schema [grounded_schema : OCNode]
(thows exception if the OCNode is of the wrong type)
```

and set via:

```
OCAtom set_predicate [grounded_predicate: OCNode] [tree : combo]
OCAtom set_schema [grounded_schema : OCNode] [tree : combo]
```

(thows exception if the OCNode is of the wrong type)

which return the atom they are called with.

Evaluation, Execution, Truth Value Computation, and Queries

What makes the OCP core come alive is the evaluation/execution of grounded procedures, the computation of truth values, and the processing of queries. These processed are invoked in Combo with the commands `evaluate`, `execute`, `compute_tv`, and `query`, described below. When combo is linked to the real core (rather than the pseudocore), inference may be invoked when truth values are computed and when queries are processed.

evaluate and execute

```
OCTruthValue evaluate [grounded_predicate : OCAtom] [arg : OCAtom]
combo execute [grounded_schema  : OCNode] [arg : OCAtom]
```

(throws an exception if called with an OCNode that is not a grounded predicate/schema).

These commands are used to explicitly call for evaluation/execution of a grounded procedure (built-in or combinator tree), on the argument given, and return the result. For example, let's say that `foo` is a combinator tree schema. Then

```
execute foo (list (cons arg1 (cons arg2 nil)))
```

is equivalent to saying

```
(get_schema foo) arg1 arg2
```

compute_tv

```
OCTruthValue compute_tv [atom : OCAtom]
OCTruthValue compute_tv [link_type : OCType] [outgoing : list<OCAtom>]
OCTruthValue compute_tv_on [table : OCAtomTable] [link_type : OCType] [outgoing :
list<OCAtom>])
```

Computing the truth value of an atom is different from calling `get_tv` because inference may be invoked, and built-in predicates and schema may be evaluated. In the first version, the atom's actual truth value may be altered. The second and third version allows the truth value of link to be computed without actually creating the link in the core. Note that to compute the truth value of a link which links to other links, the sublinks must be added to the core. Other atoms in the system may have their truth values updated as well as a result of a `compute_tv` call.

query

```
list<OCAtom> query [atom : OCAtom]
list<OCAtom> query_on [table : OCAtomTable] [atom : OCAtom]

list<list<OCAtom> > multiquery [atom : OCAtom]
```

```
list<list<OCAtom> > multiquery_on [table : OCAtomTable] [atom : OCAtom]
```

Queries allow the OCP core to be accessed like a database. The `query` and `query_on` command recursively searches their argument for occurances of a variable node. The return value is a list giving all valid assigOCents of these variables that satisfy the query (i.e., strength above a threshold). For queries with multiple variables, the `multiquery` or `multiquery_on` command must be used; in this case, the return value is a list of lists of atoms. The first list in the return value contains these varaible nodes. It is followed by lists giving all valid assigOCents of these variables that satisfy the query.

For example:

```
create_link (atom_type "InheritanceLink") (cons (concept "moshe") (cons (concept
"chicken") nil)) (simple_tv 1 1)
create_link (atom_type "InheritanceLink") (cons (concept "chicken") (cons (concept
"animal") nil)) (simple_tv 1 1)
create_link (atom_type "InheritanceLink") (cons (concept "eel") (cons (concept "animal")
nil)) (simple_tv 1 1)

multiquery (inherits $x $y) -> (($x,$y),

(#moshe:ConceptNode, #chicken:ConceptNode),

(#chicken:ConceptNode, #animal:ConceptNode),

(#eel:ConceptNode, #animal:ConceptNode))

query (inherits $x $x) -> nil

query (inherits $x (concept "animal")) -> (#chicken:ConceptNode, #eel:ConceptNode)
```

Note that in the last case, inference might conclude that since moshe inherits from chicken and chicken inherits from animal, that moshe is also an animal, and return him as well (this won't happen when using the pseudocore).

Combo Shorthands

Finally, this subsection lists shorthands, which are the cannonical ways to manipulate nodes/links when working with OCP Combo and Sasha.

Predefined Truth Values

A few truth values are predefined for convenience; `TRUE, FALSE, UNKNOWN`. Currently, these are simple truth values, defined as:

```
TRUE:=simple_tv 1 1
FALSE:=simple_tv 0 1
UNKNOWN:=simple_tv 0 0
```

Obtaining Nodes

The following are used to obtain a node of a particular type from the OCP core. Consistant with the `get_node` semantics, these functions create the atom if it doesn't already exist. Only the most commonly used node types are represented here; its very simple to create additional shorthands following the same syntax for different types.

```
//ConceptNodes
OCNode CONCEPT [name : string]
//WordNodes
OCNode WORD [name : string]

//PredicateNode (ungrounded)
OCNode PRED [name : string]
//SchemaNode (ungrounded)
OCNode SCHEMA [name : string]

//ComboTreePredicateNode
OCNode COMBO_PRED [name : string]
//ComboSchemaNode
OCNode COMBO_SCHEMA [name : string]

//BuiltInConceptNode (reflective concepts)
OCNode BUILT_IN_CONCEPT [name : string]
```

Binary Links

As a general rule, the semantics of all links in OCP are binary. The main exceptions are ListLinks, which can have any arity and represent ordered lists of arguments, and ExecutionLinks, which are trinary, and have the form `(SchemaNode,ArgumentList,Output)`. For the others cases, it is convenient to have link accessor functions that take two arguments rather than a list. Again, `get_link` semantics are used, so the link is created if it doesn't already exist.

Inheritance

Intuitively, we wish to say that Atom A inherits from Atom B, symbolized `INH A B` to the extent that:

- whatever is generally true of members of B, is also generally true of members of A
- whichever entities are members of A, are also members of B

In terms of subsets, this is

```
P(x in B || x in A)
```

Similarity

Similarity is the symmetrical analogue of inheritance,

```
P(x in A and x in B || x in A or x in B)
```

Implication and Equivalence

Logical implication and equivalence.

Membership and Subset

Set-theoretic membership (member argument comes first, then set argument), and subset.

Evaluation and Execution Output (ExOut)

Find the truth value of a predicate on an argument, or the output of a schema on an argument.

```
OCLink INH [a : OCAtom] [b : OCAtom]
OCLink SIM [a : OCAtom] [b : OCAtom]
OCLink IMPL [a : OCAtom] [b : OCAtom]
OCLink EQUIV [a : OCAtom] [b : OCAtom]
OCLink MEMBER [a : OCAtom] [b : OCAtom]
OCLink SUBSET [a : OCAtom] [b : OCAtom]
OCLink EVAL [a : OCAtom] [b : OCAtom]
OCLink EXOUT [a : OCAtom] [b : OCAtom]
```

List Links

```
OCLink LIST [l : list<OCAtom>]
```

creates a list link with the relevant atoms.

Predicates and Schema

Shorthands for applying predicates and schema are:

```
OCAtom Predicate [N : nat] [pred : OCAtom] [arg1 : OCAtom] ... [argN : OCAtom]
OCAtom Schema [N : nat] [schema : OCAtom] [arg1 : OCAtom] ... [argN : OCAtom]
```

for example,

```
compute_tv (Predicate 1 (PRED "isChicken") (CONCEPT "moshe"))
```

to compute the extent to which moshe is a chicken.

Built-in Procedures

The built-in schema `AND,OR,NOT` can be used with the shorthands

```
OCAtom AND [a : OCAtom] [b : OCAtom]
OCAtom OR [a : OCAtom] [b : OCAtom]
OCAtom NOT [a : OCAtom]
```

so for example, you can say

```
compute_tv (AND (CONCEPT "moshe") (CONCEPT "ben"))
```

to compute the truth value of "moshe and ben".

The built-in predicates `FOR_ALL,THERE_EXISTS` can be used with the shorthands

```
OCAtom FOR_ALL [variable : OCAtom] [expr : OCAtom]
OCAtom THERE_EXISTS [variable : OCAtom] [expr : OCatom]
```

(variable must be a VariableNode, expr can be a variable-laden expression)

for example, you can say

```
compute_tv (FOR_ALL $x (IMPL (Predicate 1 (PRED "isChicken") $x) (Predicate 1 (PRED
"isSilly") $x))
```

to compute the extent to which all chickens are silly.

Transformation of Complex Programs

One of the most important components of MOSES is the Reduct library, which reduces program trees to a normal form, so as to enable easier probabilistic modeling.

As of July 2008, however, Reduct does not handle reduction of program trees containing variables or equivalent representational mechanisms like general combinators. This is a shortcoming that needs to be overcome if OpenCog is to achieve its AGI goals. This page summarizes some of the literature that is relevant to this task.

Mostly the references here come from the literature on the automated program transformation of functional languages. There is plenty of work that can be ported to the Combo context, but that some in-depth reading will need to be done to figure out exactly which of the several program transformation approaches we want to use. There are several approaches with complex strengths and weaknesses, and basically all of them have been elaborated with a view toward accelerating rather than normalizing programs, which means that all of them will have to be carefully evaluated with a different view than the one their creators were taking.

Here is a pertinent quote from

http://www.cs.yale.edu/publications/techreports/tr1229.pdf

which is a good recent PHD thesis in the area

" There are numerous methods for transforming functional programming languages. In their survey paper [56], Partsch and Steinbrueggen classify various methods for program trans- formation into two basic approaches: (1) the generative set approach, which is based on a small set of simple rules which in combination are very expressive and (2) the schematic approach which is based on using a large catalog of laws, each performing a significant transformation. Fold/unfold [15] and expression procedures [69] are examples of the for- mer. The Bird-Meertens Formalism (or Squiggol) [11, 48, 49] is an example of the latter. "

In that thesis an approach called PATH is proposed, that synthesizes the generative and schematic approaches in an application to Haskell program transformation.

Assorted, Possibly Useful Approaches

On the other hand, this approach

www.ocaml.info/msc_thesis/msc_thesis.ps

automates program transformation for functional languages with strict rather than lazy typing.

This paper presents an approach to automated program transformation

http://citeseer.ist.psu.edu/10699.html

based on term rewriting rules, which is interesting (note that Luke Kaiser's Speagram interpreter is based on term rewriting internally).

This paper is also somewhat interesting:

www.cs.yale.edu/homes/tullsen/imlforpt.ps

Moshe found a nice paper called "A Tutorial on the Universality and Expressiveness of Fold", at

http://citeseer.ist.psu.edu/301385.html.

which describes the algebra of the fold operator and suggests that this might be usable as the central feature of our program transformation framework. But even if we go with this, it's not clear at the moment what the "node vocabulary" within the normalized trees would be. Fold only, or do we use other constructs, using the fold algebraic rules to derive the algebraic rules for the other constructs?

Sinot's "Director Strings As Combinators"

Most interesting, perhaps, the following paper seems to contain a workable solution to dealing with variables and combinators together ...

The paper on "efficient reduction with director strings" seems particularly relevant, and useful.

http://www.dcc.fc.up.pt/~frs/papers/sinot-jlc05-dsrev.pdf

and there is other relevant stuff on the guy's website

http://www.dcc.fc.up.pt/~frs

What he describes is a generalization of the old idea of "director strings."

This is my best bet as to the path we should take (Ben Goertzel)

Chapter Six: The Mind OS (aka the OpenCog Framework)

OpenCog As OCP's Mind OS

The foci of this OpenCogPrime wikibook are the OCP cognitive architecture and dynamics, and the general OCP software architecture. By and large, in these pages, we are intentionally avoiding discussion of OCP implementation details, which form a long, complex and subtle story in themselves. However, one lesson that we've learned in our years of experience designing and building large-scale AGI-oriented software systems like Webmind and OCP is that the mathematical/conceptual and implementation levels cannot be separated nearly as thoroughly as one might like. Just as the human mind is strongly influenced by the nature of its underlying wetware, OCP is strongly influenced by the nature of its underlying hardware and operating system. In this page and its children we review some aspects of the OCP software architecture that it seems really can't be avoided without making the conceptual material about OCP excessively vague and opaque.

OpenCogPrime:MindOS

OpenCogPrime:AutomaticParameterAdaptation
OpenCogPrime:Scheduling

OpenCogPrime:CIMDynamics OpenCogPrime:ManagingComplexAtomspaces

OpenCogPrime:AtomFreezingAndDefrosting
OpenCogPrime:DistributedCognition

OpenCogPrime:CognitiveConfiguration

Layers Intervening Between Source Code and AI Mind

There are multiple layers intervening between a body of AI source code and a conceptual theory of mind. How many layers to explicitly discuss is a somewhat arbitrary decision, but one way to picture it would be as in the following table:

	Level of Abstraction	Description/Example

1	Source code	
2	Detailed software design	
3	High-level software design/ architecture	largely programming-language-independent, but not hardware-architecture-independent: much of the material in this chapter, for example ... most of the OpenCog Framework
4	Mathematical/conceptual AI design	e.g., the sort of characterization of OCP given in most of this book
5	Abstract mathematical modeling of cognition	e.g. the SMEPH model, which could be used to inspire or describe many different AI systems
6	Philosophy of mind	e.g. the Psynet Model of Mind, patternism, etc.

Some High-level Implementation Issues

This chapter describes the basic architecture of OCP as a software system, implemented within the OpenCog Framework (OCF). We describe here those aspects of the OpenCog Framework that are most essential to OpenCog Prime. The OpenCog Framework forms a bridge between the mathematical structures and dynamics of OCP's *concretely implemented mind,* and the nitty-gritty realities of modern computer technology. All the Atoms and MindAgents of OCP live within the OpenCog Framework, so a qualitative understanding of the nature of the OCF is fairly necessary for an understanding of how the system works; and of course a detailed understanding of the OCF is necessary for doing concrete implementation work on OCP.

The nature of the OCF is strongly influenced by the quantitative requirements imposed on the system, as well as the general nature of the structure and dynamics that it must support. The large number and great diversity of Atoms needed to create a significantly intelligent OCP, demands that we pay careful attention to such issues as multithreading, distributed processing, and scalability in general. The number of Nodes and Links that we will need in order to create a reasonably complete OCP is still largely unknown. But our experiments with learning, natural language processing, and cognition over the past few years have given us an intuition for the question. We believe that we are likely to need billions — but probably not trillions, and almost surely not quadrillions — of Atoms in order to achieve a high degree of general intelligence. Hundreds of millions strikes us as possible but overly optimistic. In fact we have already run OCP systems utilizing hundreds of millions of Atoms, though in a simplified dynamical regime with only a couple MindAgents acting on most of them.

The ideal hardware platform for OCP would be a massively parallel hardware architecture, in which each Atom was given its own processor and memory. The closest thing created in the history of computing has been the Connection Machine (Hillis, 1986): a CM5 was once built with 64000 processors and local RAM for each processor. But even 64000 processors wouldn't be enough for a highly intelligent OCP to run in a fully parallelized manner, since we're sure we need more than 64000 Nodes.

Connection Machine style hardware seems to have perished in favor of more standard SMP machines. It is true that each year we see SMP machines with more and more processors on the market. However, the state of the

art is still in the *hundreds of processors* range, many orders of magnitude from what would be necessary for a *one Atom per processor* OCP implementation. And while these top-end multiprocessor SMP server machines would be very nice for a *many Atoms per processor* OCP implementation such as we have now, they are also extremely expensive per unit of memory and processing power, compared with commodity hardware.

So, at the present time, technological and financial reasons have pushed us to implement the system using a relatively mundane and standard hardware architecture. OCP version 1 will most likely live on a network of high-end commodity SMP machines. These are machines with a few Gigabytes of RAM and a few processors, so a highly intelligent OCP requires a cluster of dozens and possibly hundreds or thousands of such machines (we think it's unlikely that tens of thousands will be required, and extremely unlikely that hundreds of thousands will be). Given this sort of architecture, we need effective ways to swap Atoms back and forth between disk and RAM, and carefully manage the allocation of processor time among the various CIM-Dynamics that demand it. The use of a widely-distributed network of weaker machines for peripheral processing is a serious possibility, and we have some detailed software designs addressing this option; but for the near future we believe that this can only be a valuable augmentation to core OCP processing, which must remain on a dedicated cluster.

Of course, the use of specialized hardware is also a viable possibility — but the point is, we are not counting on it. Hugo de Garis (de Garis and Korkin, 2002) has devised techniques for using FPGA's (Field-Programmable Gate Array) to implement very efficient evolutionary learning of 3D formal neural networks; and it may be that such techniques could be modified to yield efficient evolutionary learning of OCP procedures. More speculatively, it might be possible to use evolutionary quantum computing to accelerate OCP procedure learning. Or, in a different direction, Octiga Bay has created an intriguing supercomputer architecture that integrates both FPGA's and a special processor interconnect fabric that provides extremely rapid cross-processor networking — this might well make the overhead involved with distributed OCP processing considerably less. All these possibilities are exciting to envision, but the OCP architecture does not require any of them in order to be successful.

Marvin Minsky is on record [NOTE: it would be nice if someone could dig up the reference for this, I know I read such an interview somewhere!! — Ben G] expressing his belief that a human-level general intelligence could be implemented on a 486 PC, if we just knew the algorithm. We doubt this is the case, and it is certainly not the case for OCP. By current computing hardware standards, a OCP system is a considerable resource hog. And it will remain so for at least several years to come, considering Moore's Law.

It is one of the jobs of the OCF to manage the system's gluttonous behavior. It is the software layer that abstracts the real world efficiency compromises from the rest of the system — this is why we call it a "Mind OS": it provides services, rules, and protection to the Atoms and CIM-Dynamics that live on top of it, which are then allowed to ignore the software architecture they live on.

We will not discuss the coding details of the OCF here, but we will sketch a few of the more important particulars of the design. Our goal here is to illustrate some of the more implementation-oriented aspects of OCP in a platform and programming-language-neutral mathematical way, and to show that the implementation of these formalisms is both necessary and possible. We will address four major aspects of the engineering side of OCP development:

- Distribution of Atoms and CIM-Dynamics across multiple machines and functional units.
- Scheduling of processor time (or as we say, attention allocation) to CIM-Dynamics.
- Caching of unimportant Atoms to disk (freezing) and their reloading onto RAM upon demand (defrosting).

- Automatic control of the system's numerous parameters, to prevent it from crashing.

These concepts will be refined in the following two chapters, when we enumerate the system's core Dynamics and discuss multiple possible OCP Configurations.

What we will describe here is basically the OCF architecture as intended for OCP in its "human level AGI-capable version," with comments occasionally inserted in places where the current OCF/OCP software architecture differs from this (due to time-constraints during implementation or other short-term practical reasons). For instance, at time of writing (July 2008) the OCF implementation does not support distributed processing nor Atom-by-Atom freezing/defrosting. It has been coded so as to make extension to distributed processing reasonably simple (e.g. Links don't involve lists of Node objects, they involve lists of Node handles, where a handle is an object that may be resolved to a local Node or to a Node on a remote machine). It does do freezing/defrosting but in a cruder way than desirable, involving saving/loading large groups of Atoms at a time. Automated parameter adaptation is currently implemented in a relatively simplistic way; the Webmind AI Engine, a predecessor to the NCE, contained a full parameter adaptation subsystem very much along the lines described here.

Varieties of Atomspace

In other pages reviewing hypergraph formalism and terminology (MindOntology: SMEPH, AtomFormalization), we introduced one kind of Atomspace, which here will be called the Simple-Atomspace: simply, a container of Atoms. Conceptually, we don't need anything besides that. However, to accurately mathematically model what goes on in the MindOS (in a complex, distributed OCF instance), we need more refined and complicated spaces. We need to handle distributed processing, in which an Atomspace may be spread among multiple units. And we need to handle caching, in which the Atoms in an Atomspace may be in different states regarding their availability in the cache.

To deal with distributed processing, we must introduce further notions such as

```
Multipart-Atomspace = Union_k  (Multipart-Atomspace Union Simple-Atomspace)^k
```

(where ^*k* means the k'th power according to the Cartesian product).

A Multipart-Atomspace is an Atomspace that consists of a number of distinct parts. The definition allows for nesting — Multipart-Atomspaces that live inside Multipart-Atomspaces — but this nesting is only allowed to go on for a finite number of levels, because all sets are assumed finite. In practice we do not go beyond two levels of recursion, in the current OCP design. That is, we have a couple cases of Multipart-Atomspaces that contain Multipart-Atomspaces; but it goes no further.

In practice the separate parts of a Multipart-Atomspace will usually be functionally specialized parts, devoted to one or another particular cognitive task.

A caching Atomspace embodies a distinction between parts of an Atomspace that are active in RAM, and parts that have been saved to disk. We may define

```
Caching-Atomspace = Atomspace^3
```

where this structure has the following interpretation. A OCP that is capable of caching and reloading (*freezing* and *defrosting*) atoms may be considered, at any given time, as a triple

```
(live atoms, frozen proxy atoms, frozen atoms)
```

The frozen atoms are on disk, the live atoms are in RAM, and the frozen proxy atoms are shells living in RAM pointing to frozen atoms on disk, existing only so that live atoms can use them as handles to grab frozen atoms by.

Real Atomspaces may combine the features of caching and distributed processing. There are two simple ways to do this, global caching where there is one cache for the whole distributed network, and local caching where each part of the system has its own cache. There are also more complex possible arrangements, in which some parts share a cache and others have their own, giving rise to many combinations. The two simpler cases are covered by:

```
Globally-Caching-Multipart-Atomspace = (Multipart-Atomspace x Atomspace)^2

Locally-Caching-Multipart-Atomspace = Union_k Caching-Atomspace^k
```

A globally caching multipart Atomspace, is a multipart Atomspace with one cache for the whole system; a locally caching multipart Atomspace has a different cache for each part.

In general a Multipart-Atomspace that has caches associated with various groupings of its parts may be called a Complex-Atomspace, a category that groups together the various possibilities mentioned above.

Dynamics

So we have a set of Atoms, distributed across machines, cached as necessary. Now how does this Atomspace evolve?

In the Webmind AI Engine (a predecessor of the NCE), we gave Nodes and Links the freedom to receive attention directly from the CPU, and then decide what actions to take. This was consistent with Webmind's Multi-agent System architecture and design, and it was also satisfyingly close to the conceptual foundations of the patternist perspective on mind. However, it led to inefficiency as the number of actors in the system scaled up. In order to address this performance issue, in architecting OCP, we decided it was necessary to create a software system that deviated further from the structures naturally suggested by the philosophical foundations. This doesn't mean that the OCP architecture fails to represent the philosophical foundations; it means rather that the mapping from philosophy to software is a little more complex. This complexity is the price paid for efficient performance on contemporary commodity hardware.

In OCP, most of the Atoms are inert, in the sense that they don't contain pieces of code telling them how to act on other Atoms. (The exceptions are grounded SchemaNodes and PredicateNodes, which correspond to OpenCogPrime:ComboTrees living in the OpenCogPrime:ProcedureRepository; but in this case, there is a MindAgent that chooses nodes of this type and enacts their internal schemata/predicates for them) . Rather than cycling through Atoms and letting them explicitly act, as was done in Webmind, OCP dynamics works using dynamics objects called MindAgents. On each machine implementing part of a OCP Atomspace, there is a set of MindAgents, which are repeatedly cycled through, each getting a certain amount of processor time, and each

using its processor time to *enact* actions on behalf of certain selected Atoms. Most of the MindAgents in the software system represent CIM-Dynamics — dynamical processes that are concerned with modifying and creating Atoms in an Atomspace — but there are also MindAgents that take care of purely system-level tasks rather than mind-ish tasks.

The interrelation between the conceptual/mathematical structure of Atomspaces, and the machines implementing the Atomspaces, is moderately subtle. A multipart Atomspace, as defined above, refers to an Atomspace that is conceptually divided into several different parts. But each of these different conceptual parts may be implemented on one or more machines. So, for instance, a simple Atomspace may be running on one, three, or 47 different machines. A MindAgent running on a certain machine has free access to all Atoms within its containing simple Atomspace. Some MindAgents may choose to primarily act on Atoms living in the same machine as they do. But if need be, they can freely act on Atoms living on other machines serving the same Atomspace. On the other hand, if an Atom A is on a machine living in a different Atomspace from MindAgent M, then it can't be assumed that M has the power to modify A. Interactions between different simple Atomspaces in the same multipart Atomspace are handled by the OpenCogPrime:MindDB design. Essentially, the MindDB represents a centralized database of Atoms, and mediates cases where different Units want to affect each others' Atoms, or independently contain Atoms that represent the same thing (i.e. are *copies* of each other).

So, when a CIM-Dynamic-embodying MindAgent is active, it selects Atoms (by some criterion, often importance) from the Atomspace in which it lives, and acts on them, modifying their state and/or creating new Atoms based on them. For instance, there is one MindAgent that deals with schema execution, which selects SchemaInstanceNodes and allows them to enact the schema functions within themselves, thus transforming other Atoms or creating new ones. There is another that deals with first-order logical inference, inspecting logical links in the Atomspace and creating new ones.

Processor time allocation among MindAgents is done based on a simple scheduling process — different types of action, embodied in different MindAgents, receive different slices of time, and then they can allocate their slices using different policies. We will return to this scheduling algorithm in OpenCogPrime:Scheduling.

We use the term OpenCogPrime:Unit to refer to a simple Atomspace together with a collection of CIM-Dynamics (which in practice are implemented in MindAgents). Each Unit may potentially run on multiple machines; we use the term Lobe to refer to an individual software process that is part of a Unit. A Unit consists in general of multiple Lobes running in multiple software processes (generally on multiple machines), but sharing the same "distributed virtual Atomspace." Most discussions regarding OCP cognition may be carried out on the level of Units rather than Lobes, but for instance the discussion of MindAgent scheduling in OpenCogPrime:Scheduling is an exception.

Next, we use the term OpenCogPrime:UnitGroup to refer to a multipart Atomspace, each component of which is associated with some CIM-Dynamics, and which as a whole may also be associated with a collection of CIM-Dynamics. In theory a Unit group could be a grouping of groupings of groupings of ... groupings of Atomspaces. In the current OCP design it never gets all that deep. For instance, we have the Evolutionary Programming Unit-Group, which contains a set of Unit-Groups corresponding to evolving populations with different fitness functions. And in a full-on self-modification OCP configuration, we will have Unit-Groups corresponding to whole OCP systems, each being studied and optimized by a *global controller* OCP contained in its own Unit-Group. This results in a maximum depth of three: So, we have, in the worst case that seems

feasibly likely to be necessary for human-level AGI, a Unit-Group that is a set of sets of sets of simple Atomspaces.

A little more formally, if we define

```
Complex-Dynamic-AtomSpace = ComplexAtomspace x CIM-Dynamic-Set
```

then we see that the above discussion implies the definition

```
Unit = Complex-Dynamic-Atomspace

Unit-Group = Union_k Unit^k
```

We may then define

```
Dynamic-Multipart-Atomspace = Union_k (Unit-Group Union Unit)^k
```

A complex OCP configuration, at any given time, is a OpenCogPrime:ComplexAtomSpace: that is, a dynamic multipart Atomspace. It will contain multiple Units, which host different configurations of CIM-Dynamics, and contain a subset of the Atoms in the OCP system.

System Parameters and Control

One important property of OCP is that it is highly configurable via adjustment of a large number of parameters.

This page discusses how parameters are handled in OCP, and how the OCF automatically controls them to prevent such problems.

Some have argued that any AGI system with a large number of parameters, on which the system's behavior potentially depends in a sensitive way, is necessarily doomed to fail. However, this seems an unrealistically idealistic perspective. The human brain, if modeled as a mathematical dynamical systems, contains an incredible number of adjustable parameters, and if any of them gets outside their acceptable range, the functionality of the overall system may be dramatically impacted. Most mind-altering drugs operate via making relatively small tweaks to the levels of specific chemicals in the brain, thus inducing subjectively massive alterations in overall system state. Nudging some parameter of some brain-process 10% out of line may cause madness or death. The truth is that living systems rely on multiple quantitative parameters remaining within their acceptable ranges, and contain complex parameter interdependencies that guide overall system functionality in such a way that when one parameter threatens to move too far out of line, the others adjust themselves in such a way as to mitigate the threat.

While it's valuable to work to minimize the number of parameters in OCP, and especially to minimize the number of parameters on which the system's dynamics depends sensitively, and to minimize the number of parameters whose direct and obvious impact extends beyond a particular subsystem of the system — in reality, this can never be done completely, and so it is necessary to have systems in place to enable automated parameter adaptation, so that OCP can behave like a biological system in terms of keeping its own critical parameters within acceptable bounds, and (the next step) to some extent auto-adjusting its own parameters to improve its efficiency at achieving its goals.

Architecturally, parameter optimization is handled in the current OCP design by three MindAgents:

- HomeostaticParameterAdaptation MindAgent, which adapts parameters according to certain rules that it contains, every system cycle
- ParameterAdaptationRuleLearning MindAgent, which learns the rules used by the HPAMindAgent
- LongTermParameterAdaptation MindAgent, which optimizes parameters to maximize system-wide goal-achievement, in a slow way based on sophisticated analysis

As of July 2008 none of these are implemented yet.

System Parameters

What are these parameter values, that these parameter-adaptive MindAgents deal with? Each CIM-Dynamic has a number of parameters, some of which may vary between different instances of the same dynamic that are active in the same OCP system at the same time. These system parameters are complicated in nature. Of the parameter set for a CIM-Dynamic type, we may say that:

- Some are under control of that CIM-Dynamic type, and will be the same for all instances across all Units.
- Some are under control of the hosting Unit, and will vary across Units but will be the same for all instances in a single Unit.
- Some are under control of the particular instance.

Parameters may be considered as members of the abstract space Parameter-Space, and may be specified by a quadruple:

```
(controller, type, range, value)
```

where

- controller *within* {CIM-Dynamic, Unit, instance}
- type *within* {Float, Integer, Boolean, String}
- range = list of values, if type equals (Boolean or String)
- range = interval, if type equals (Float or Integer)
- value *within* range

The type and range are the same for a parameter across all Units in a OCP instance. The value can be set by the controller, inside the valid range. Thus, the parameter set of a OCP instance with k types of CIM-Dynamics, j Units and n particular instances of those dynamics can be defined as

```
CIM-Dynamic-Parameter-Set = Union_k Parameter-Space

Unit-Parameter-Set = Union_j Parameter-Space

Instance-Parameter-Set = Union_n Parameter-Space

OCP-Parameter-Set = CIM-Dynamic-Parameter-Set x

Unit-Parameter-Set x Instance-Parameter-Set
```

Some semi-random, hopefully evocative examples of important parameters for OCP:

- k, the default personality parameter of the inference engine
- The rate of importance decay for each kind of Node and Link.
- The maximum sizes or expected sizes of compound PredicateNodes and SchemaNodes, during learning processes.

Homeostasis

We have seen that the values of the multiple system parameters can influence system behavior very drastically. This influence has two main aspects, which we call intelligence and health.

The influence of parameter values on intelligence can be seen when incorrect parameters cause the system to make wrong inferences (the k parameter mentioned above), or waste resources (by scheduling its CIM-Dynamics in sub-optimal ways), for example.

The influence of parameter values on system health has more serious consequences. When wrong parameters influence the system's health, it can crash, as we've already mentioned, or it can proceed very slowly due to spending all its resources on unnecessary OCF-level tasks.

Ideally, we would expect an intelligent system to be able to tune its own parameters, for maximum health and intelligence. The former is easier than the latter. This section presents our approach to homeostatic control, the OCF component responsible for the system's health. Automated parameter tuning for increased intelligence is an aspect of intelligent, goal-directed self-modification, discussed in a later chapter.

The health of a OCP system can be measured by a number of HealthIndicators, which are formally the same as FeelingNodes, but have a different orientation than other FeelingNodes. A HealthIndicator embodies a formula that assesses some aspect of the system's state, and the overall feeling of Health is then a combination of HealthIndicator values. Maximizing Health should be one of the system's conscious goals, therefore the overall Satisfaction FeelingNode that comprises one of the systems ubergoals (top-level goals) should contain Health as a component. However, at least in the early stages of development, the system will not want to rely on its explicit goal-achieving function to ensure Health, and some Health-maintenance rules may be hard-coded into the system, inside MindAgents dealing with "homeostatic control."

Homeostatic control (HC), as a general process, acts by keeping track of the system's health and the past values of each HealthIndicator. HC acts in two modes: long term optimization and firefighting. The two modes are handled by different MindAgents.

Firefighting

Firefighting mode is executed by the HomeostaticParameterAdaptation MindAgent. This MindAgent checks, each system cycle, if any of the system's HealthIndicators have come close to a dangerous value. This can happen when the system is running out of memory, or when the time it takes to answer queries from user applications becomes too long, or when the amount of new knowledge generated by the system's cognitive dynamics drops too low, for example.

In this case, the HC will identify the problematic HealthIndicators, and change the parameters responsible for these indicators. How does it know which changes need to be made? This is a combination of a set of built-in rules with automated mining of past information.

The built-in rules are hard-coded into the system, and are variations of the form:

```
if HealthIndicator > threshold change Parameter by amount
```

An example would be: *if the system's free memory is below x%, increase i1 by y%.* The parameters x and y would then be tuned by experimentation. Multiple rules may be created to address the same problem. The above rule will cause more Atoms to be removed from RAM and frozen. However, it may be that the rate at which new Atoms are being generated is so fast that this wouldn't solve the problem. In that case, the system can activate other rules that can:

- Increate the rate of importance decay for all Atoms
- Decrease the rate of Atom creation for some CIM-Dynamics

Therefore, the HC should keep track of the rules it has applied in the recent past to address a given HealthIndicator, so it can use alternatives when it fails to mitigate the risky situation in the first attempt.

Coding enough rules to fight all possible fires would be a complex task. Not only does one need a large number of rules — a bigger problem is that the correct set of rules may be very difficult to identify. OCP has numerous parameters, and numerous health indicators. Since each parameter can influence multiple health indicators, there is ample space for emergent behavior in the space of HealthIndicators, and changes in the parameters can have results that are very hard to predict. Also, different OCPs with different Atomspaces may react differently to the same rules.

The solution is to hard-code only very rigorous rules, ones that will definitely prevent the system from crashing, but at a potentially large cost. Other rules have to be learned.

The process by which this learning takes place is the mining of temporal information, which will be discussed in detail later on. Here, a brief outline will suffice. The system keeps track of changes done to its parameters, and measure how they have affected HealthIndicators. This information is stored in a DB that's optimized for the following operation.

When faced with a potentially dangerous situation, HC will look in the DB for situations in which the HealthIndicators had values close to the present ones, which were followed by healthier outlooks. It will then identify the differences in the parameter space, and apply those differences.

This is a continuously improving process; the more experience the system has, the better it will behave under these circumstances. A critical mass is necessary, if this method is to work well, and this will be provided under controlled circumstances during the early life of a new OCP, being part of Experiential Learning.

Long term optimization

Complementary to firefighting mode, we have the LongTermParameterAdaptation MindAgent. This is the slow mode in which the HC process is always seeking to improve the system's health, over a very long timescale. It does so by changing system parameters on purpose, on a controlled environment. When resources are available, the LTPA MindAgent will allocate one or more Units for this process, where the effects of these changes can be observed without any harm to the other parts of the system. This is a very elementary kind of introspection.

This speculative experimentation, and its careful monitoring, will immediately provide more data for the data mining based firefighter. It will also slowly allow the system to crystallize knowledge about good and bad values for each system parameter. This process will eventually give rise to a more proactive HC function, which will alter the range of valid parameter values according to this long term learning system.

Scheduling: Allocating Attention Among MindAgents

We now discuss in detail how processor time is allocated in OCP, within a single Unit.

At the highest level, processors are allocated to OpenCogPrime:UnitGroups and OpenCogPrime:Units.

One level down from there, processor time is given within each OpenCogPrime:Lobe (remember, multiple OpenCogPrime:Lobes comprise a OpenCogPrime:Unit)

- to CIM-Dynamics, implemented within each Lobe as MindAgents, which then select Atoms and act on them
- to OpenCogPrime:Tasks, spawned by MindAgents and sometimes by other core processes

So, at the OpenCogPrime:Lobe level, attention allocation in OCP has two aspects: how MindAgents and Tasks receive attention from OCP, and how Atoms receive attention from different CIM-Dynamics. The topic of this page is the former. The latter is dealt with elsewhere, in two ways:

- in the chapter on OpenCogPrime:AttentionAllocation, which discusses the dynamic updating of the AttentionValue structures associated with Atoms, which determine how much attention various MindOntology:FocusedCognitiveProcesses (embodied in OpenCogPrime:CIMDynamics) pay to them
- in the discussion of various specific CIMDynamics, each of which may make choices of which Atoms to focus on in its own way (though generally making use of AttentionValue and TruthValue in doing so)

The Attention Allocation subsystem is also pertinent to MindAgent scheduling, because it discusses dynamics that update ShortTermImportance (STI) values associated with MindAgents, based on the usefulness of MindAgents for achieving system goals. In this page, we will not enter into such cognitive matters, but will merely discuss the mechanics by which these STI values are used to control processor allocation to MindAgents.

We have multiple kinds of CIM-Dynamics, which can be grouped in innumerable ways among different Units. Multiple instances of the same CIM-Dynamic can co-exist in the system, either in the same or in different Units. One can think of a CIM-Dynamic as a class in an object-oriented system, which can be instantiated multiple times, with different properties each time. (Each instantiation is a particular MindAgent running on a particular machine within a particular Unit.) This gives us several options, including:

1. Allocate attention by the type of CIM-Dynamic, and then sub-allocate attention for each instance.
2. Allocate attention directly to each CIM-Dynamic instance, in a single step.

While the former option would give us finer-grained control of the system, we feel that the latter is both more cognitively natural and more efficient. Therefore, we have chosen to attach an AttentionValue object to each CIM-Dynamic, and use it to schedule processor time. Each CIM-Dynamic then has an STI value (along with possibly other AttentionValue components), which is meaningful in the context of its hosting Unit.

An Example Scheduler Design

Each Lobe within a Unit has a Scheduler, which allocates slices of processor time to the MindAgents hosted in that Unit. At any given time, the Scheduler is enacted when a processor becomes available, and it has to decide which MindAgent will become active next. The simplistic approach is to keep a pool of inactive MindAgents, and select one from the pool, with a probability proportional to its normalized STI. However, things aren't that simple, and this heuristic fails to take into consideration a number of factors:

- Attention needs to be given proportionally to the importance of each CIM-Dynamic. The importance of the CIM-Dynamic is altered dynamically by the Unit and by other CIM-Dynamics (a topic to be discussed later on, under the general rubric of *economic attention allocation*)
- Starvation has to be prevented — even the least important CIM-Dynamic needs to receive some attention from the system.
- In SMP machines, multiple CIM-Dynamics will be active at any given time. Some dynamics will not behave properly if executed at the same time as other dynamics.

This suggests that a moderately complex scheduling algorithm is required.

In this section we describe one such algorithm, which was implemented in the NCE. It is not adequate for long-term OCP usage and the current (July 2008) OCP code contains something even simpler that is also not adequate for long-term usage. This page will be updated with more details regarding scheduler designs in the hopefully near future. Note that scheduling is a fairly mature area of computer science due to its importance in the design of operating systems for various types of devices, so this is not an area where new ideas need to be invented for OCP; rather, designs need to be crafted meeting OCP's specific requirements based on state-of-the-art knowledge and experience.

In the example scheduler design, there are two important inputs:

- An exclusion list for each MindAgent, listing the MindAgents that it has incompatibilities which. Incompatible MindAgents can't be run concurrently.
- The STI associated with each MindAgent.

In the current NCE implementation, the Scheduler maps the MindAgent importances to a set of priority queues Q, and each queue is run a number of times per cycle. The number of queues is a parameter, and the function that maps a queue's priority level to the number of executions is:

```
executions-per-cycle = priority-level
```

where i is usually 1. In possession of the number of times each MindAgent should be executed, the Scheduler builds a binary matrix, in which each column corresponds to one MindAgent, and each row corresponds to one step in the cycle. Two MindAgents are set to true in the same row only if they can be run concurrently.

This matrix drives scheduling for that cycle. It is compacted through merging of rows with compatible restrictions (the optimal solution for this minimization problem is very expensive to find, but simple heuristics work well), to reduce overhead. Threads are assigned to the MindAgents and activated/put to sleep as the Scheduler goes through the steps.

When the importance of a CIM-Dynamic or other MindAgent changes, one just has to add/remove a few true flags in the matrix, which is a cheap operation. When MindAgents are added or removed, one has to rebuild the whole matrix, which is more expensive, but still doable, as it's not a very frequent operation inside a Unit.

Finally, Task execution is currently handled via allocating a certain fixed percentage of processor time, each cycle, to executing the top Tasks on the queue. Adaptation of this percentage may be valuable in the long term but was not yet implemented in NCE.

Concretely-Implemented Mind Dynamics

The "cognitive dynamics" in a OCP system may be considered on two levels:

1. the dynamics explicitly programmed into the source-code (these are the Concretely-Implemented Mind Dynamics, or CIMDynamics)
2. the dynamics that are intended to emerge through the system's self-organizing dynamics, based on the cooperative activity of the CIMDynamics on the shared Atomspace

Most of the material in these Wiki pages has to do with the particular CIMDynamics in the OCP system. In this section we will simply give some generalities about the CIMDynamics as abstract processes and as software processes, which are largely independent of the actual AI contents of the CIMDynamics.

In practice the CIMDynamics involved in a OCP system are fairly stereotyped in form, although diverse in the actual dynamics they induce.

As noted above, CIMDynamics could be coded in Combo and represented as grounded SchemaNodes, but in the current codebase they're hardcoded as C++ objects, for reasons of efficiency and (secondarily) programmer-convenience.

We return here to the trichotomy of cognitive processes presented in Cognitive Architecture Overview, in which OCP cognitive processes may be divided into:

- MindOntology:ControlProcesses
- MindOntology:GlobalCognitiveProcesses
- MindOntology:FocusedCognitiveProcesses

In practical terms, these may be considered as three categories of CIMDynamic.

Control Process CIMDynamics are hard to stereotype. Examples are:

- the process of "homeostatic parameter adaptation" of the parameters associated with the various other CIMDynamics
- the CIMDynamics concerned with the execution of schemata in the ActiveSchemaPool

Control Processes tend to focus on a limited and specialized subset of Atoms or other entities, and carry out specialized "mechanical" operations on them (e.g. adjusting parameters, interpreting schemata). To an extent, this may be considered a "grab bag" category containing CIMDynamics that are not global or focused cognitive processes according to the definitions of the latter two categories. However, it is a nontrivial observation about

the OCP system that the CIMDynamics that are not global or focused cognitive processes are all explicitly concerned with "control" in some sense.

Global and Focused Cognitive Process CIMDynamics all have a common aspect to their structure. Then, there are aspects in which Global versus Focused CIMDynamics diverge from each other in stereotyped ways. (And finally, of course, there are aspects in which particular CIMDynamics differ from each other, or they would not be different!)

In most cases, the process undertaken by a Global or Focused CIMDynamic involves two parts: a selection process and a main process. Schematically, such a CIMDynamic typically looks something like this:

1. Grab a bunch of Atoms that it is judged will be useful to process, according to some selection process
2. Do something with these Atoms, possibly together with previously grabbed ones (this is what we sometimes call the *main process* of the CIM-Dynamic)
3. Go back to step 1

Of course, this way of breaking down dynamics is, like caching and multipart Atomspaces, an artifact of the implementation of mind on von Neumann hardware. In a massively parallel system one would still have CIMDynamics but their construction would be different than what's described here, even if the AI effect were basically the same. Webmind 2000 used a somewhat different framework for structuring (its analogue of) CIMDynamics, because it tried to more closely mimic how CIMDynamics would be implemented on a parallel machine.

The major difference between Global and Focused cognitive processes lies in the selection process. In the case of a Global process, the selection process is very broad, typically yielding a whole AtomTable, or a very significant subset of one. On the other hand, in the case of a Focused process, the selection process is very narrow, yielding only a small number of Atoms, which can then be processed more intensively (and expensively, on a per-Atom basis).

Selection Processes

Formally, selection processes may be considered as part of the space of mappings

```
Selection-Process [ Atomspace –> Atomspace]
```

That is, they take in a large Atomspace (for instance an AtomTable, in the implementation), and output a potentially smaller Atomspace (the set of Atoms selected).

The selection processes we use now, in practice, can be understood as compositions of simpler selection processes of various types.

First, we have selection processes that may be termed high-level restrictions. For example:

- The selection process that selects only the *live* atoms from a MultipartAtomspace
- The selection process that selects only the atoms in a particular part of a MultipartAtomspace
- The selection process that selects only the *live* or *proxy* atoms from a MultipartAtomspace
- The selection process that selects only the *live* atoms in a particular part of a MultipartAtomspace

High-level restrictions make their choices based only on which part of a caching structure, and/or which part of a multipart structure, an atom lives in.

Then we have type-based restrictions, selection processes that pick only atoms of certain types (e.g. only nodes, or only binary Links, or only WordNodes, etc.) The key thing is that a type-based restriction makes its choices based only on relationship type or node type.

The above sorts of selection processes are used by both Global and Focused cognitive processes. That is, a cognitive process considered Global may still restrict itself to only live Atoms, or to only live Links, etc. The key point is that a Global process is iterating through a whole AtomTable or a huge fraction thereof and carrying out some cheap process on each Atom it visits — so its selection processes tend to be pretty simplistic and surface-level, not related to Atom "contents."

Then we have selection processes that may be termed fitness-oriented selectors, which are utilized by Focused cognitive processes. For example:

- The process that picks one atom from a set with a probability based on some numerical quantity associated with the atom (e.g elements of TruthValue or AttentionValue)
- The process that picks two atoms from a set, independently, with the probability of each choice based on some numerical quantities associated with that individual atom (e.g. elements of TruthValue or AttentionValue)

There are also more specific selection processes, which choose for example Atoms obeying some particular combination of relationships in relation to some other Atoms; say choosing only Atoms that inherit from some given Atom already being processed. There is a notion, described in the PLN book, of an *Atom Structure Template* — this is basically just a predicate that applies to Atoms, such as

```
P(X).tv
```

equals

```
( (Inheritance X cat) AND (Evaluation eats(X,cheese) ).tv
```

which is a template that matches everything that inherits from *cat* and eats cheese. Templates like this may be used to select Atoms from the AtomTable, which gives a much more refined selection than the above, more generic sorts of selection process.

All the SelectionProcesses in use in OCP now are obtained by composing a fitness-oriented selector with a type-based restriction with a high-level restriction. Let's call these standard selection processes. And all the CIMDynamics in use in OCP right now may be understood as composing Schema with standard selection processes.

What CIMDynamics Do With Selected Atoms

So, what do practical CIMDynamics look like in OCP today? As noted above, the *meat* of the CIMDynamics will be described in following chapters: they are a diverse bunch, as befits the richness of the emergent cognitive processes to which they are designed to collectively give rise.. In terms of their use of standard

selection processes, however, some generalizations are possible. One way to say it is that there is a selection process that chooses some Atoms, and then a main process that takes these Atoms output by the selection process as input and

- Sometimes gives other Atoms as output.
- Sometimes modifies some of the numerical parameters associated with the input Atoms
- Sometimes deletes the input Atoms

Tasks

The next point to be made is more software architecture oriented. It is not convenient for OCP to do all its work directly via the action of MindAgent objects embodying CIMDynamics. In the NCE we found it necessary to augment the MindAgent framework with an additional scheduling mechanism that we call the Task framework. In essence, this is a ticketing system, designed to handle cases where MindAgents or Schema spawn one-off tasks to be executed — things that need to be done only once, rather that repeatedly and iteratively as with the things embodied in MindAgents. This does not yet exist in OCP as of July 2008, but it will need to.

For instance, *grab the most important Atoms from the Atomspace and do shallow PLN reasoning to derive immediate conclusions from them* is a natural job for a MindAgent. But *do search to find entities that satisfy this particular predicate P* is a natural job for a Task. When a Task is created it is submitted to the TaskScheduler and then put in a queue behind any other Tasks with higher priority.

What one sees here is that, in the current implementation, practical issues have pushed us into making a number of distinctions that are not conceptually fundamental. We distinguish between MindAgents, Tasks and Schema; and in the current (July 2008) codebase, MindAgents and Tasks embody their operational code in separately-coded C++ rather than including internal schemata. When we move to a system in which MindAgents and Tasks contain pointers to SchemaNodes inside (which will be possible once we create a more efficient schema execution framework) then the definitions of MindAgent and Task will refer purely to scheduling. A MindAgent will be defined as a Schema that is executed by the system in every system cycle, and a Task will be defined as a Schema that has been put in the Tasks queue.

The Management of Complex Atomspaces

As we have seen, complex OCP configuration involves a collection of interacting "units". Most of these units are dynamic simple Atomspaces. A handful, however, may be dynamic complex Atomspaces themselves, containing simple Atomspaces as components. So far we have discussed the way CIM-Dynamics act within a simple Atomspace. Now we turn to some issues regarding the management of complex Atomspaces. There are three sorts of issues here:

- How do different Units and Unit-Groups interact?
- How do the machines inside a multi-machine Unit manage the Unit-local distributed processing
- How does the system overall regulate the distribution of resources among various Units and Unit-Groups?

Here we will give an overview of our treatment of these issues. Further details have been discussed and documented separately, but we omit them here, as these issues have more to do with distributed processing than with AI. This sort of work is not easy to do, but it is a well understood part of computer science, and while

OCP's requirements as a distributed system are subtle, they are not outside the scope of the existing theory and practice of distributed systems design and implementation.

Control and Distributed Processing

In general, we may assume that a multi-Unit OCP is regulated by a process called a MultipartAtomspaceController or MAC, which is not an Unit but rather a piece of software designed to regulate sets of Units. There may sometimes be the need to introduce some non-Unit servers into a OCP, carrying out specialized auxiliary algorithms. These servers will also be regulated by the MAC.

One has the problem of physically starting up, shutting down and monitoring a bunch of machines in a Webmind. This is also done by the MAC. Note that the MAC in the current framework must regulate all the machines in the OCP, regardless of which Unit they're assigned to.

Within a distributed Unit, one has the problem of load balancing. This means that the Atoms in the Unit may be redistributed among the various machines assigned to the Unit, in order to maximize the effectiveness of the system. In practice one wishes to maximize the frequency with which CIM-Dynamics get to act, and minimize the amount of messaging between the machines in the Unit. This is an optimization problem which we have worked out in detail, but which we will not discuss here.

Distribution and Live Caching of Atoms in OCP

Since we can't fit all of OCP's Atoms in a single machine, and we want to minimize communication between machines, how do we handle the distribution of the Atoms? Currently, OCP uses a combination of global and local caching Atomspaces — a design that is centered around the concept of a *Mind DB*.

The Mind DB based design is founded on a few simple assumptions:

- OCP has multiple, specialized units.
- Each unit carries out a number of CIM-Dynamics on an Atomspace that contains a subset of the global Complex-Atomspace.
- Some Atoms may be used in different units, by different CIM-Dynamic;

The latter assumption implies that different transformations may be effected upon these Atoms by the different CIM-Dynamics. This suggests that we need a special CIM-Dynamic to reconcile conflicting changes. The solution that most appealed to us was to create a specialized Unit which is responsible for being the central Atom storage. This is the Petaverse.MindDB.

The Petaverse.MindDB is then a Globally-Caching-Atomspace and an associated set of CIM-Dynamics. These are the dynamics responsible for freezing and defrosting of Atoms (as examined in the previous section), and for the reconciliation of conflicting transformations on a single Atom (using PLN rules such as revision and the Rule of Choice), as well as the more mundane tasks of transaction control and data distribution. All the Atoms that should be in the Atomspace available to the OCP mind as a whole are included in the Mind DB, in one of the three possible states already mentioned.

In order to minimize messaging between machines, one may want to introduce MindDBProxies in each machine. A MindDBProxy contains local copies of Atoms that live inside the Petaverse.MindDB. This is a way

of dealing with the frequent situation where there are numerous Atoms that are heavily used by multiple CIM-Dynamics, across multiple Units. By keeping proxies of them distributed around, one minimizes the amount of inter-machine messaging significantly, thus speeding up the whole system.

Freezing and Defrosting

In the OCP design, attention is given to Atoms only by specific dynamics embodied in MindAgents or Tasks, which then allow Atoms to carry out particular operations. However, as we've hinted in the introduction, the number of Atoms in a Novamente system is very large, and not all Atoms can be realistically kept in memory all the time. This is the reason for the creation of the cache-enabled Atomspaces.

When one enables caching, only a subset of Atoms is kept in RAM (usually a reasonably small subset), and the other Atoms are cached out to disk — we say they're frozen, because they can't be the subjects of any transformations made by the CIM-Dynamics.

Atoms are frozen (cached to disk) and defrosted (loaded back into RAM) based on a component of their AttentionValue called the Long-Term Importance (LTI). When an Atom's LTI drops below a certain threshold i1, it is removed from RAM, and a proxy is kept in its place. The proxy uses much less space than the original Atom, and it remains involved in the relationships that included the cached Atom.

Maintaining the proxy allows the system to keep the Atom's LTI value current, through the process of Importance Updating, which is described in the page on AttentionAllocation. When the proxy's LTI increases above a second threshold i2, the system understands that the Atom has become relevant again, and loads it from the Cache.

Eventually, it may happen that the proxy doesn't become important enough over a very long period of time. In this case, the system should remove even the proxy, if its Long Term Importance (LTI) is below a third threshold i3. Other actions, usually taken by the system administrator, can cause the removal of Atoms and their proxies from RAM. For instance, in a Novamente system managing information about a number of users of some information system, the deletion of a user from the system would cause all that user's specific Atoms to be removed.

When Atoms are saved to disk and have no proxies in RAM, they're considered totally frozen, and can only be reloaded by the system administrator. When reloaded, they will be disconnected from the rest of the Atomspace, and should be given special attention in order to pursue the creation of new Links with the other Atoms in the system.

It's important that the values of i1, i2, and i3 be set correctly. Otherwise, one or more of the following problems may arise:

- If i1 and i2 are too close, the system may spend a lot of resources with freezing and defrosting.
- If i1 is set too high, important Atoms will be excluded from the system's dynamics, decreasing its intelligence.
- If i3 is set too high, the system will forget very quickly and will have to sped resources re-creating necessary but no longer available evidence.

Generally, we want to enforce a degree of hysteresis for the freezing and defrosting process. What we mean is that:

```
i2 - i1 > c1, c1 > 0
i1 - i3 > c2, c2 > 0
```

This ensures that when Atoms are defrosted, their importance is still above the threshold for freezing, so they will have a chance to build new Links and become more important, and won't be frozen again very quickly. It also ensures that frozen Atoms stay in the system for a period of time before their proxies are removed and they're definitely forgotten.

Distributed Cognition

This page discusses distributed cognitive processing in OCP. The topic is just lightly reviewed — it is a large and deep topic, but one more to do with "standard computer science" than with AGI per se. Getting these difficult sorts of standard computer science done right is critical for achieving AGI on current networks of von Neumann machines, but, does not require dramatic original conceptual innovations going far beyond the current state of the art.

Specialized Strategies for Distributing Particular Cognitive Algorithms

There are many ways to set up distributed processing in OCP, consistent with the above generalities regarding complex Atomspaces. The following section addresses this issue on a high level — it reviews the way a multipart Atomspace can be divided into functionally distinct *units*, each corresponding to a particular domain of cognition, such as language processing, visual perception, abstract reasoning, etc. But there are also specifics of distributed processing that correspond to cognitive algorithms rather than domains of cognition. The two main cognitive algorithms in OCP are PLN and probabilistic evolutionary learning (PEL), and each of these naturally lends itself to different styles of distribution. Each functionally specialized unit as described in the following chapter may then consist of a network of machines, including subnetworks carrying out distributed PLN or distributed PEL according to special distribution architectures appropriate for those algorithms. A detailed description of approaches for distributing PLN or PEL would bring us too far afield here, but we will give a rough indication.

In the case of PLN, one useful "trick" may be to take N nodes and partition them among K machines, and then use, on each machine, only the links between the nodes on that machine. There are then cycles of inferencing and repartitioning. This allows inference to deal with a body of knowledge much larger than can fit on any one machine, but without any particular inference MindAgent ever needing to wait for the Mind OS to make a socket call to find the Atom at the end of a Link.

On the other hand, distributed PEL is an even simpler beast, conceptually. Evolutionary algorithms like GA/GP, BOA and MOSES lend themselves very naturally to distributed processing — if one is evolving a population of N items (e.g. SchemaNodes), then one can simply partition them among K machines and speed up fitness evaluation by a factor of roughly K (subtracting off a little bit for communication overhead, but in the usual case of expensive fitness functions this isn't a significant matter). The subtler aspect of distributed PEL is distributing the probabilistic modeling and instance generation phase. This can also be done, though less simply than distributing fitness evaluation, but the details depend on which species of probabilistic modeling one is

using inside BOA. For instance, if an expensive approach for instance generation is used, then the probabilistic model of fit population elements can be distributed across several machines and each machine can generate a subset of the instances required.

Globally Distributed Processing

Finally, we must not omit the possibility of broadly distributed processing, in which OCP intelligence is spread across thousands or millions of machines networked via the Internet. Even if none of these machines is exclusively devoted to OCP, the total processing power may be massive, and massively valuable.

In terms of OCP core formal structures, a globally distributed network of machines, carrying out peripheral processing tasks for a OCP system, is best considered as its own, massive, Multipart-Atomspace. Each computer or local network involved in carrying out peripheral OCP processing is itself a simple Atomspace. The whole Multipart-Atomspace that is the globally distributed network must interact with the Multipart-Atomspace that is the primary OCP system — or perhaps with multiple separate Multipart-Atomspaces representing several OCP systems drawing on the same pool of Webworld locations.

The use of this kind of broadly distributed computing resource involves numerous additional control problems, of course and we will not address these here. A simple case is massive global distribution of PEL fitness evaluation. In the case where fitness evaluation is isolated and depends only on local data, this is extremely straightforward. In the more general case where fitness evaluation depends on knowledge stored in a large Atomspace, it requires a subtler design, wherein each globally distributed PEL subpopulation contains a pool of largely similar genotypes, and contains a cache of relevant parts of the Atomspace, which is continually refreshed during the fitness evaluation process. This can work so long as each globally distributed lobe has a reasonably reliable high-bandwidth connection to a machine containing a large Atomspace.

Cognitive Configuration

This page discusses "cognitive configuration," or the particular use of the distributed computing architecture provided by the Mind OS to enable effective partitioning of cognitive processes across multiple Units.

The need for cognitive configuration is rooted in a familiar philosophical point: the difficulty for pragmatic AGI posed by the limited computational capability of realizable substrates for intelligence. The assumption of massive processing power renders the AI problem trivial. Limitations in processing power require practical intelligent systems to adopt a host of heuristics that constrain generality and flexibility but permit reasonably flexible general intelligence together with effective specialized intelligence in selected domains. And, one reflection of this notion of generalization/specialization balance is the need for functional specialization. In practical terms, what this means is that intelligent systems will generally be divided into parts, each part being concerned with some particular sort of task, some particular aspect of intelligence. In abstract terms, this can be viewed in the SMEPH framework in terms of cluster hypergraphs.

A partitioned hypergraph is simply a hypergraph that is approximately divided into sub-hypergraphs, with the property that, on average, each node has a lot more nontrivial-strength connections to other nodes within its sub-hypergraph than to nodes outside it. This is a purely structural notion. A cluster hypergraph, then, is a partitioned hypergraph in which the set of vertices associated with each partition are roughly clustered together.

In general, clustering-together may be measured in terms of any metric defined on the set of vertices. So there tend to be a lot of "nearby" nodes within a partition, and not so many between partitions.

In a SMEPH context, the metric is the Jaccard metric associated with SimilarityLinks between ConceptEdges and SchemaEdges. So in a clustered SMEPH hypergraph, each partition, roughly, deals with a distinct sort of static and dynamic patterns.

Hypergraph clustering may occur on many different levels. The notion of *functional specialization* as we'll discuss it here has to do mostly with the highest level of clustering — with the division of an intelligent system into a set of modules dealing with distinct sorts of declarative and procedural knowledge. But the same notion also applies on finer levels of granularity.

As an example of functional specialization, the human brain is roughly divided into regions dealing with language processing, regions dealing with visual perception, regions dealing with temporal perception, regions dealing with motor control, and so forth. The consequence of this brain-level partitioning is that the derived hypergraph of the brain is correspondingly partitioned — and the structures and dynamics of the ConceptEdges and SchemaEdges corresponding to the different regions of the brain tend to cluster together (language Concepts/Schemata in one partition, vision Concepts/Schemata in another partition, etc.).

The key reason why this heuristic of functional specialization is valuable for intelligence is that solving hard problems is often rendered easier by hierarchical modular problem decomposition. Functional specialization is basically the implication, on the whole-intelligent-system level, of the simple problem-solving heuristic of breaking a hard problem down into components, solving each component, and then piecing together the componentwise solutions to form a whole solution. This heuristic doesn't work for all hard problems, of course. But we conjecture that if there's a hard problem this doesn't work for, then it's incredibly unlikely that an intelligence will evolve to solve this problem (because evolution tends to create somewhat modular solutions), and it's also unlikely that human beings will design a system to solve the problem (because humans naturally think in terms of modular-breakdown heuristics).

The term we have adopted for the construction of appropriate functionally specialized sub-units (analogous to functionally specialized brain regions) is cognitive configuration. The OCP design reflects the need for cognitive configuration directly, via the notion of ComplexAtomspaces. In a complex OCP configuration, each part of a multipart Atomspace is supplied with MindAgents and Atoms that cause it to correspond to a certain specialized domain — which, assuming the dynamics unfolds as intended, causes the derived hypergraph of the system to take the form of a SMEPH cluster hypergraph.

More technically, in terms of the OCP design, cognitive configuration has several aspects:

- How the overall Atomspace is divided into parts (assuming a multipart Atomspace, the general case)
- What types of Atoms are initially placed into each part
- What particular Atoms (from what data sources) are placed into each part
- Which of the general CIM-Dynamics are placed into each part, and what parameters they're given
- What specialized new CIM-Dynamics, if any, are placed into each part (this comes up only occasionally)

All this is "CIM configuration" as distinct from underlying hardware configuration. There are relations between CIM configuration and underlying hardware configuration, but they're not strict ones; the same CIM

configuration may be realized hardware-wise in many different ways. Generally speaking, each pat of the multipart Atomspace will correspond either to a single machine or to a tightly-connected cluster of machines.

In addition to the main reason for functional specialization mentioned above — the general power of the hierarchical-decomposition heuristic — there are also other reasons for constructing a OCP with a complex cognitive configuration, instead of just a big, teeming, self-organizing Atomspace.

The first reason is related to hardware configuration. Messaging between machines in a distributed-computing system is relatively slow, and if we can break much of the business of mind down into a collection of Units many of which can operate on individual machines, then — given the realities of von Neumann hardware — we'll have a more efficient OCP.

The second reason is purely cognitive in nature. Each collection of CIM-Dynamics has its own emergent nature as a dynamical system. The dynamics of two loosely connected Units is not going to be the same as the dynamics one would get if one threw all the Atoms and CIM-Dynamics from the two Units into the same big pool. In some cases, it seems, the *loosely connected* dynamics are more cognitively desirable. We will consider some examples of this below. For instance, it may be useful to have a Unit in which inference acts very speculatively, and another one in which it acts very conservatively. Mixing these two up will result in a Unit in which inference acts moderately speculatively — but consistently moderately speculative inference does not generally lead to the same results as the combination of conservative and highly speculative inference. In fact, there is ample psychological evidence indicating that it is important to creativity for the mind to have some kind of partially isolated subsystem in which anything goes.... (see Ben Goertzel's book "From Complexity to Creativity").

The basic cognitive configuration to be used for an embodied learning focused OCP is described in the overall architecture diagram in the Introduction. Elaboration of aspects of the configuration described there is given in later chapters, where for instance the cognitive arguments underlying the AttentionalFocus Unit are described (very loosely, this corresponds to the notion of "working memory" in the human mind: it is a collection of Atoms that are marked as extremely important and deserving of intensive processing on a pool of dedicated machines).

In the remainder of this section we discuss a couple critical ways in which the cognitive configuration described in the Figure may be extended. Some potential extensions are fairly basic, such as:

- A Psynese Unit, for dealing with interactions with other OCPs using the Psynese language, a scheme that's been developed for separate OCP systems to communicate via exchanging sets of Atoms directly, rather than by exchanging information in English, Lojban or other human languages
- A "speculation" Unit, similar to the GlobalAttentionalFocus Unit, but with parameters of CIM-Dynamics specifically tuned for the creation of potentially interesting low-confidence Links and hypothetically interesting Nodes.

Others are more involved and subtle.

Multiple Interaction Channels

The architecture diagram given in CognitiveArchitecture is somewhat simplified in that it shows a single "interaction channel" for interaction with the outside world. This is correct for a OCP instance that is

specialized to control a single agent in a simulation world but is not the most general case. One big difference between humans and OCPs is that, in principle, a OCP can carry on a large number of separate interactions at the same time — for instance, a single OCP could control a large number of agents in various different simulation worlds, along with a number of physical robots and some disembodied chatbots. The human brain is not designed for this kind of multitasking but OCP, with a proper cognitive configuration, can be. In OCP lingo, we describe this difference by saying that a OCP can have several interaction channels, not just one like humans have.

Each interaction channel (InteractionChannels), in an interaction-configured OCP, should come along with a dedicated AttentionalFocus. There may also be a need for additional CIM-Units corresponding to different sensation and action modalities. For instance, in an embodied-agent-control scenario, an InteractionChannel would correspond to a single agent being controlled. If a single OCP instance were controlling several agents, this would correspond to multiple InteractionChannels. But a single InteractionChannel may wind up dealing with a number of qualitatively different modalities, such as vision, hearing, smell, rolling on wheels, arm movement, and noisemaking. Some of these modalities may be sufficiently complicated as to require a whole specialized Unit on their own, feeding into the channel-specific AttentionalFocus (as opposed to simpler modalities which may just feed data directly into the channel-specific AF). The prototypic example of a complex sensory modality is vision. Action-wise, for instance, rolling on wheels clearly doesn't require its own Unit, but merely the execution of simple hard-wired schemata; but, moving legs with complexity comparable to human legs may require enough complexly coordinated decisions to merit its own Unit.

A channel-specific AttentionalFocus Unit is dedicated to rapid analysis of things that are important as regards the particular interaction channel in question. This is a different CIM-Unit from the global AF of the whole system, because what's important in the context of the current conversation isn't necessarily what's important to the long-term goals of the system. On the other hand, there is still a need for a part of the mind that embodies a deep and thoroughgoing quasi-real-time integration of all mental processes: which is the global AF. The AF bifurcation enables the system to respond in a rapid way to perceptual stimuli (including conversations), without interrupting systematic long-term thought about important things. In the human mind there's a constant tension between (the analogues of) interaction-specific AF and global AF, but there's no need to port this kind of confusion into the OCP design.

Self-Modification Configuration

Next, what if we want a OCP that can modify its own CIM-Dynamics? This is the first step toward a deeply, fully self-modifying OCP. This is provided in the architecture diagram by a "control schema learning" Unit — but in the most general case, things become more complex than that. To enable maximally powerful and general self-modification, we need a configuration that is a step up in abstraction from what we've been discussing so far. We need to introduce the notion of a UnitSet — a set of Units that work together, each forming a coherent group carrying out cognitive processes in a collective way.

Define a schema-adaptive configuration as follows:

- A collection of UnitSets, each one devoted to carrying out some set of standard cognitive tasks
- A process devoted to controlling these experimental Units: feeding them new cognitive schema and new tasks, and recording their performance
- An Unit devoted to thinking about candidate cognitive schema to try out in the population of evolving UnitSets

- An AttentionalFocus Unit devoted to thinking hard about extremely good candidate cognitive schema to try out in the population of evolving UnitSets

This is a generic self-modification-oriented configuration, devoted to the learning of new cognitive schema.

Now, in a schema-adaptive configuration, the CIMDynamics are fixed. The next step is to do something similar for CIM Dynamics. One may define a CIMDynamic-adaptive configuration as

- A collection of UnitSets, each one devoted to carrying out some set of standard cognitive tasks
- A process devoted to controlling these experimental Units: feeding them new CIMDynamics and new tasks, and recording their performance
- A Unit devoted to thinking about candidate CIMDynamics to try out in the population of evolving UnitSets
- An AttentionalFocus Unit devoted to thinking hard about extremely good candidate CIMDynamics to try out in the population of evolving UnitSets

One may define a fully adaptive configuration similarly, except that each UnitSet in the population is not merely adapting CIMDynamics, it is adapting other aspects of its sourcecode. This requires recompilation of each experimental UnitSet at the time it is launched. Of course, at this stage, one may arrive at a new UnitSet whose evolved code implies discarding the whole framework of UnitSets, evolutionary programming, and so forth — although it is unlikely that this will be an early-stage result.

Finally, it is clear that if one wanted to take the next step and have the system modify its own source code, the same kind of architecture could be used. Instead of experimental Units, one would have experimental OCPs, and the controller process would monitor and launch these OCPs. Analysis of the results of experimentation, and creation of new candidate OCPs, would proceed basically as in the picture sketched above. One *merely* has a learning problem that's an order of magnitude more difficult. Clearly, it makes sense to have the CIM-Dynamics of the current (or a near descendant) OCP implementation optimized as far as possible through self-modification, before turning this optimized OCP to the much harder task of modifying its own source.

By this point, we have gone rather far out into speculative-design-space, relative to the current state of practical OCP experimentation. However, we believe it's important to go this far out at the design phase, in order to determine, with reasonable confidence, whether one's AGI design is in principle capable of going all the way. We believe that the OCP design is sufficiently flexible and sufficiently powerful to support full, radical self-modification. We know exactly what OCP structures and dynamics will be useful for radical self-modification, what configuration will be required, and so forth. The path ahead is fairly clearly visible, as are the various obstacles along it.

Use of Globally Distributed Processing

The role of globally distributed computing in OCP's quest for intelligent self-modification is a supporting one, rather than a central one, but is worth briefly noting nonetheless. The evolution of ensembles of slightly varying OCP systems is something that must take place on powerful dedicated machines, at least in the relatively near future. However, much of the schema learning that is involved in self-modification can be farmed out to a population of weaker machines. For instance, suppose the system wants to learn a better deductive reasoning rule; this may involve a great deal of empirical testing of candidate deductive reasoning rules on datasets, which is a kind of operation that is easily farmed out to a globally distributed network.

Chapter Seven: Integrative Inference

Integrative Inference

- Adaptive Inference Control
- Inference Pattern Mining
- Hebbian Inference Control
- Incorporating Other Cognitive Processes Into Inference
- Confidence Decay

The PLN inference framework is not discussed extensively in this wikibook because it has a whole other, paper book devoted to it, *Probabilistic Logic Networks*. However, that book is about PLN rather than OCP, and doesn't go into depth on the integration of PLN with OCP.

The purpose of this page and its children is to discuss issues specifically pertinent to the integration of PLN with other OCP processes. Some such issues are dealt with in other pages already, such as OpenCogPrime:EvolutionBasedInferenceControl which discusses PEL as a form of inference control. The issues not dealt with in other pages, are gathered here.

In the main the issues discussed here pertain to inference *control* rather than the nature of individual inference steps. This is because individual inference steps are the same in PLN whether PLN is being utilized within or outside of OCP. On the other hand, inference control is done within OCP in ways that wouldn't necessarily make sense in a standalone-PLN context. The main theme of the chapter is adaptive inference control, but this is only a viable option if there is some method besides PLN that is studying patterns among inferences, and/or studying inference-relevant patterns among Atoms. If one were doing complex PLN inference outside the context of an integrative AI framework containing appropriate non-inferential cognitive processes, one would need to take a quite different approach. In fact, our suggestion is that integrative AI is essentially the only workable approach to the control of higher-order inference in complex cases. Certainly, attempts to solve the problem with a narrow-AI approach have been made many times before and have not succeeded.

Types of PLN Query

The PLN implementation in OCP is complex and lends itself for utilization via many different methods. However, a convenient way to think about it is in terms of three basic backward-focused query operations:

- findtv, which takes in an expression and tries to find its the truth value
- findExamples, which takes an expression containing variables and tries to find concrete terms to fill in for the variables
- createExamples, which takes an expression containing variables and tries to create new Atoms to fill in for the variables, using *concept creation* heuristics as discussed in a later chapter, coupled with inference for evaluating the products of concept creation

and one forward-chaining operation:

- find Conclusions, which takes a set of Atoms and seeks to draw the most interesting possible set of conclusions via combining them with each other and with other knowledge in the AtomTable

These inference operations may of course call themselves and each other recursively, thus creating lengthy chains of diverse inference.

(In an earlier version of PLN, these operations actually had the above names. I am not sure if these precise command names have survived into the current implementation, but the same functionality is there. Maybe this page should be updated.)

Findtv is quite straightforward, at the high level of discussion adopted here. Various inference rules may match the Atom; in our current PLN implementation, loosely described below, these inference rules are executed by objects called Evaluators. In the course of executing findtv, a decision must be made regarding how much attention to allocate to each one of these Evaluator objects, and some choices must be made by the objects themselves — issues that involve processes beyond pure inference, and will be discussed later in this chapter. Depending on the inference rules chosen, findtv may lead to the construction of inferences involving variable expressions, which may then be evaluated via findExamples or createExamples queries.

The findExamples operation, on the other hand, sometimes reduces to a simple search through the Atomspace, as in the test examples above. On the other hand, it can also be done in a subtler way. If the findExamples Evaluator wants to find examples of $X so that F($X), but can't find any, then its next step is to run another findExamples query, looking for $G so that

```
Implication $G F
```

and then running findExamples on G rather than F. But what if this findExamples query doesn't come up with anything? Then it needs to run a createExamples query on the same implication, trying to build a $G satisfying the implication.

Finally, forward-chaining inference (findConclusions) may be conceived of as a special heuristic for handling special kinds of findExample problems. Suppose we have K Atoms and want to find out what consequences logically ensue from these K Atoms, taken together. We can form the conjunction of the K Atoms (let's call it C), and then look for $D so that

```
Implication C $D
```

Conceptually, this can be approached via findExamples, which defaults to createExamples in cases where nothing is found. However, this sort of findExamples problem is special, involving appropriate heuristics for combining the conjuncts contained in the expression C, which embody the basic logic of forward-chaining rather than backward-chaining inference.

A Toy Example

To exemplify the above ideas, I will now give a detailed *log* of a PLN backward chainer as applied to resolve simple findtv and findExamples queries over a small test database, consisting of a few dozen relationships

regarding a half-dozen people. The example shows the actual output from a prior version of the OCP/PLN testing framework that we used to qualitatively assess PLN performance, in mid-2005. The style of output from the current version differs (due to the greater generality of the current version): at the moment we no longer use the English-like output that was used in the prior version, because it become confusing in the context of highly complex examples.

In this example we begin with the findtv query

```
findtv friendOf(Amir,Osama)
```

The heuristic then searches in the knowledge base to find what relationships are known involving the terms in the query. It first finds the following relationship involving friendOf, which indicates the (probabilistic, not certain) symmetry of the friendOf relationship:

```
friendOf is symmetricRelation (0.65,0.72)
```

Note that symmetry is stored as a relationship like any other. This allows PTL to deal nicely with the cases of relationships that are only partially, probabilistically symmetric, such as friendOf and enemyOf.

The heuristic then searches for relations involving symmetricRelation, finding the following relationship, an Equivalence relationship that embodies the definition of symmetry

EQUIVALENCE TO IMPLICATION

```
if R000 is symmetricRelation then if and only if R000(X007,X008) then R000(X008,X007)
(0.995,0.985)
```

(Here the notation such as X008 refers to VariableNodes).

The system then applies this definition to the relationship friendOf, which involves a step of deduction:

VARIABLE INSTANTIATION

```
if friendOf is symmetricRelation then if and only if friendOf(X007,X008) then
friendOf(X008,X007) (0.995, 0.985)
```

DEDUCTION (*NODE TRUTH VALUE*)

```
if and only if friendOf(X007,X008) then friendOf(X008,X007) (0.654, 0.713)
```

Now it applies a step of variable instantiation, seeking to match the new link it has learned regarding friendOf with the other terms that were there in the original query.

VARIABLE INSTANTIATION

```
if and only if friendOf(Osama,Amir) then friendOf(Amir,Osama) (0.654,0.713)
```

This gives it a way to assess the truth value of friendOf(Amir, Osama). Namely: it now realizes that, since friendOf is symmetric, it suffices to evaluate friendOf(Osama, Amir). Thus it submits a findtv query of its own creation.

DEDUCTION (*NODE TRUTH VALUE*)

```
The truth value of friendOf(Amir, Osama) is unknown:
findtv friendOf(Osama, Amir)
no answer
```

But unfortunately, this query finds no answer in the system's knowledge base. The exploitation of the symmetry of friendOf was a dead end, and the inference heuristic must now backtrack and try something else. Going back to the start, it looks for another relationship involving the terms in the original query, and finds this one, which indicates that friendOf is transitive:

```
friendOf is transitiveRelation (0.4, 0.8)

if and only if R001 is transitiveRelation then if exists X010 such that
AND(R001(X009,X010),R000(X010,X011)) then R001(X009,X011) (0.99,0.99)
```

EQUIVALENCE TO IMPLICATION

```
if R001 is transitiveRelation then if exists X010 such that
AND(R001(X009,X010),R000(X010,X011)) then R001(X009,X011) (0.995,0.985)
```

VARIABLE INSTANTIATION

```
if friendOf is transitiveRelation then if exists X010 such that
AND(friendOf(Amir,X010),friendOf(X010,Osama)) then friendOf(Amir, Osama) (0.995,0.985)
```

DEDUCTION (*NODE TRUTH VALUE*)

```
if exists X010 such that AND(friendOf(Amir,X010),friendOf(X010,Osama))
then friendOf(Amir, Osama) (0.410, 0.808)
```

In this case, we end up with a findExamples query rather than a findtv query:

FIND INSTANCES

```
findExamples AND(friendOf(Amir,X010),friendOf(X010,Osama))

X010={Britney}

friendOf(Amir, Britney) (0.8, 0.3)

friendOf(Britney, Osama) (0.7, 0.6)
```

To resolve this find query, it must use the truth value formula for the AND operator

AND RULE

```
AND(friendOf(Amir, Britney),friendOf(Britney,Osama)) (0.56,0.3)
```

Since this evaluation yields a reasonably high truth value for the find query, the system decides it can plug in the variable assignment

```
X010={Britney}
```

Along the way it also evaluates

```
exists X010 such that AND(friendOf(Amir,X010),friendOf(X010,Osama)) (0.56,0.3)
```

And, more pertinently, it can use the transitivity of friendOf to assess the degree to which Amir is a friend of Osama:

DEDUCTION (*NODE TRUTH VALUE*)

```
friendOf(Amir, Osama) (0.238, 0.103)
```

There is nothing innovative or interesting about the inference trajectory followed here — the simple inference control heuristic is doing just what one would expect based on the knowledge and inference rules supplied to it. For dealing with a reasonably large-sized knowledge base, more sophisticated inference control heuristics are needed, because a simplistic strategy leads to combinatorial explosion.

What is important to observe here is how the probabilistic truth values are propagated naturally and sensibly through the chain of inferences. This requires the coordination of a number of different PLN inference rules with appropriately tuned parameters.

Because the knowledge base is so small, the revision rule wasn't invoked in the above example. It's very easy to see how it would come up in the same example with a slightly larger knowledge base, however. If there were more knowledge about the friends of Amir and Osama, then the system could carry out a number of inferences based on the symmetry and transitivity of friendOf, and revise the results together at the end.

PLN and Bayes Nets

Next, some comments on the relationship between PLN and Bayes Nets may be useful. We have not yet implemented such an approach, but it may well be that Bayes Nets methods can be a useful augmentation to PLN for certain sorts of inference (specifically, for inference on networks of knowledge that are relatively static in nature).

We can't use standard Bayes Nets as the primary way of structuring reasoning in OCP because OCP's knowledge network is loopy. The peculiarities that allow standard Bayes net belief propagation to work in standard loopy Bayes nets, don't hold up in OCP, because of the way you have to update probabilities when you're managing a very large network in interaction with a changing world, so that different parts of which get different amounts of focus. So in PLN we use different mechanisms (the "inference trail" mechanism) to avoid

"repeated evidence counting" whereas in loopy Bayes nets they rely on the fact that in the standard loopy Bayes net configuration, extra evidence counting occurs in a fairly constant way across the network.

However, when you have within the AtomTable a set of interrelated knowledge items that you know are going to be static for a while, and you want to be able to query them probabilistically, then building a Bayes Net (i.e. "freezing" part of OCP's knowledge network and mapping it into a Bayes Net) may be useful. I.e., one way to accelerate some PLN inference would be:

1. Freeze a subnetwork of the AtomTable which is expected not to change a lot in the near future
2. Interpret this subnetwork as a loopy Bayes net, and use standard Bayesian belief propagation to calculate probabilities based on it

This would be a highly efficient form of "background inference" in certain contexts. (Note that this requires an "indefinite Bayes net" implementation that propagates indefinite probabilities through the standard Bayes-net local belief propagation algorithms, but this is not problematic.)

Adaptive Inference Control

In the previous subsection we have seen an example of what the PLN inference rules can do when applied to a small test database. They can draw plausible conclusions based on the knowledge at hand. But of course, inference on a small test database is quite different from real-world inference. There is little doubt that in principle PLN is capable of complex real-world inference, in the sense that there exist meaningful chains of PLN inferences leading from the data at an embodied AGI system's disposal to the conclusions the system needs to draw to achieve its complex goals. The real question, however, is how the meaningful inference-chains are found from amidst the vast mass of meaningless ones. This is the essence of the question of *inference control.*

It is clear that in humans, inference control is all about context. We use different inference strategies in different contexts, and learning these strategies is most of what *learning to think* is all about. One might think to approach this aspect of cognition, in the OCP design, by introducing a variety of different inference control heuristics, each one giving a certain algorithm for choosing which inferences to carry out in which order in a certain context. However, in keeping with the *integrated intelligence* theme that pervades OCP, we have chosen an alternate strategy for PLN. We have one inference control scheme, which is quite simple, but which relies partially on structures coming from outside PLN proper. The requisite variety of inference control strategies is provided by variety in the non-PLN structures such as

- OpenCogPrime:HebbianLinks existing in the AtomTable.
- Patterns recognized via pattern-mining in the corpus of prior inference trails

Mechanics of the Inference Control Process

We will now describe the basic "inference control" loop of PTL in OCP. We will discuss it in the context of backward chaining; the case of forward chaining is very similar.

Given an expression to evaluate via inference (according to any one of the query operations mentioned above), there is a collection of Evaluator objects that matches the expression.

First, each Evaluator object makes a preliminary assessment regarding how likely it is to come up with decent results. This assessment is made based on some preliminary explorations of what Atoms it might use to draw its conclusions — and study of various links (including HebbianLinks) that exist relating to its actions on these Atoms (we will give some examples of this shortly), and information regarding what Evaluators have been useful in what prior inferences in related contexts (stored in a structure called the InferencePatternRepository, to be discussed below).

Then, the overall evaluation process looks at the assessments made by each Evaluator and decides how much computational resource to allocate to each Evaluator. This we may call the "Evaluator choice problem" of inference control.

Finally, each Evaluator chosen, then needs to make choices regarding which Atoms to use in its inference — and this choice must be made by use of existing links, and information in the InferencePatternRepository. This is the "Atom choice problem" of inference control.

As an example of the choices to be made by an individual Evaluator, consider that to evaluate (Inheritance A C) via a deduction-based Evaluator, some collection of intermediate nodes for the deduction must be chosen. In the case of higher-order deduction, each deduction may involve a number of complicated subsidiary steps, so perhaps only a single intermediate node will be chosen. This choice of intermediate nodes must also be made via context-dependent probabilities. In the case of other Evaluators besides deduction, other similar choices must be made.

So the basic inference algorithm we have discussed is basic backward-chaining inference, but aggressive and adaptive pruning using HebbianLinks and other existing knowledge is done at every stage.

The Evaluator Choice Problem as a Bandit Problem

The evaluator choice problem, as described above, is an example of a "multi-armed bandit problem" as commonly studied in statistics. The atom choice problem is also a bandit problem. Both of these problems can be approached via using standard "bandit problem" heuristics.

The paradigm bandit problem involves a slot machine ("multi-armed bandit") with N levers to pull, each of which may have a different odds of yielding a reward each time it is pulled. The player's goal is to maximize her earnings, but when she starts out playing with the bandit, she has no idea which levers lead to which levels of reward. So, at any given point, she must balance two factors:

- Exploitation: pulling the level that seems to give maximum reward, based on experience so far
- Exploration: trying out various levers to get a sample of their performance, so as to get more confident estimates of the reward level offered by each lever

Obviously, as more experience is gained, the bias should shift from exploration towards exploitation. There is a substantial body of mathematics regarding bandit problems, but most of the results prove inapplicable in real-world situations due to making inappropriate statistical assumptions. In practice, the two most common algorithms for solving bandit problems are:

- epsilon-greedy: spend 1-epsilon of your time exploiting the best option found so far (with "best" defined in terms of expected reward), and epsilon of your time randomly exploring other options

- softmax: assign a probability to each option, using a heuristics formula based on thermodynamics that assigns higher probabilities to options that have proved more successful in the past, with an exponential scaling that favors successful options nonlinearly over unsuccessful ones

The only modification we choose to make to these simple algorithms in the OCP context is to replace the probability with a product

```
probability * weight_of_evidence
```

This evidence-weighted probability may be used within both epsilon-greedy and softmax.

Choosing an Evaluator for an inference step within PLN is a bandit problem, where prior experience is used to provide initial probabilities for the options (which are possible Evaluators rather than levers to pull), and then inferences done using each option are used to provide increasingly confident probabilities for each option. The default approach within OCP is softmax, though epsilon-greedy is also provided and may prove better in some contexts.

In Atom selection, the options (levers) are Atoms to use within an inference rule (an Evaluator).

It is important to note, however, that the statistical nature of the Atom-choice bandit problem is different from the statistics of the Evaluator-choice bandit problem, because there are not that many Evaluators, but there are a lot of Atoms. It would be possible to merge the two into a single bandit problem, whose options were (Evaluator, Atom-set) tuples, but this seems an inferior heuristic approach. The Evaluator bandit problem involves a relatively small number of options, which makes it more tractable. So we have chosen to break down the inference control process into two levels of bandit problems: Evaluator choice and Atom choice.

The complexity of the inference control problem can be well-understood by observing that each individual step poses two difficult statistical inference problems, one nested within the other! What allows pragmatic inferences to be achievable at all is, clearly, prior knowledge, which allows most of the bandit problems occurring in a complex inference to be "pre-solved" via assumption of prior probabilities. Normally, only a handful of difficult steps in an inference need to actually be studied statistically via numerous iterations of the epsilon-greedy or softmax algorithms. On the other hand, the first few inferences in an unfamiliar domain may not connect with the system's knowledge base of prior probabilities, and may thus need to be done in a much slower, more statistically thorough way.

Inference Pattern Mining

Among the data used to guide the solution of the Evaluator choice problem, the most important component is explicit information regarding which Evaluators have been useful in which contexts during past inferences.

This information is stored in OCP in a data repository called the InferencePatternRepository — which is, quite simply, a special "data table" containing inference trees and patterns recognized therein. An "inference tree" refers to a tree whose nodes are Atoms (generally: Atom-versions), and whose links are inference steps (so each link is labeled with a certain Evaluator).

Note that, in many cases, PLN creates a variety of exploratory inference trees internally, in the context of doing a single inference. Most of these inference trees will never be stored in the AtomTable, because they are

unconfident and may not have produced extremely useful results. However, they should still be stored in the InferencePatternRepository. Ideally one would store all inference trees there. In a large OCP system this may not be feasible, but then a wide variety of trees should still be retained, including mainly successful ones but also a sampling of unsuccessful ones for purpose of comparison.

The InferencePatternRepository may then be used in two ways:

- An inference tree being actively expanded (i.e. utilized within the PLN inference system) may be compared to inference trees in the repository, in real time, for guidance. That is, if a node N in an inference tree is being expanded, then the repository can be searched for nodes similar to N, whose contexts (within their inference trees) are similar to the context of N within its inference tree. A study can then be made regarding which Evaluators and Atoms were most useful in these prior, similar inferences, and the results of this can be used to guide ongoing inference.
- Patterns can be extracted from the store of inference trees in the InferencePatternRepository, and stored separately from the actual inference trees (in essence, these patterns are inference subtrees with variables in place of some of their concrete nodes or links). An inference tree being expanded can then be compared to these patterns instead of, or in addition to, the actual trees in the repository. This provides greater efficiency in the case of common patterns among inference trees.

A reasonable approach may be to first check for inference patterns and see if there are any close matches; and if there are not, to then search for individual inference trees that are close matches.

Mining patterns from the repository of inference trees is a potentially highly computationally expensive operation, but this doesn't particularly matter since it can be run periodically in the background while inference proceeds at its own pace in the foreground, using the mined patterns. Algorithmically, it may be done either by exhaustive frequent-itemset-mining (as in the Apriori or relim algorithms), or by stochastic greedy mining. These operations should be carried out by an InferencePatternMiner.

Hebbian Inference Control

One aspect of inference control is Evaluator choice, which is based on mining contextually relevant information from the InferencePatternRepository (see OpenCogPrime:InferencePatternMining). But, what about the Atom choice aspect? This can in some cases be handled via the OpenCogPrime:InferencePatternRepository as well, but it is less likely to serve the purpose than in the case of Evaluator choice. Evaluator choice is about finding structurally similar inferences in roughly similar contexts, and using them as guidance. But Atom choice has a different aspect: it is also about what Atoms have tended to be related to the other Atoms involved in an inference, generically, not just in the context of prior inferences, but in the context of prior perceptions, cognitions and actions in general.

Concretely, this means that Atom choice must make heavy use of HebbianLinks. The formation of HebbianLinks will be discussed in the following chapter, on attention allocation. Here it will suffice to get across the basic idea. The discussion of HebbianLinks here will hopefully serve to help you understand the motivation for the HebbianLink formation algorithm to be discussed later. Of course, inference is not the only user of HebbianLinks in the OCP system, but its use of HebbianLinks is somewhat representative.

The semantics of a HebbianLink between A and B is, intuitively: In the past, when A was important, B was also important. HebbianLinks are created via two basic mechanisms: pattern-mining of associations between

importances in the system's history, and PLN inference based on HebbianLinks created via pattern mining (and inference) Thus, saying that PLN inference control relies largely on HebbianLinks is in part saying that PLN inference control relies on PLN. There is a bit of a recursion here, but it's not a bottomless recursion because it bottoms out with HebbianLinks learned via pattern mining.

As an example of the Atom-choices to be made by an individual Evaluator in the course of doing inference, consider that to evaluate (Inheritance A C) via a deduction-based Evaluator, some collection of intermediate nodes for the deduction must be chosen. In the case of higher-order deduction, each deduction may involve a number of complicated subsidiary steps, so perhaps only a single intermediate node will be chosen. This choice of intermediate nodes must be made via context-dependent prior probabilities. In the case of other Evaluators besides deduction, other similar choices must be made.

The basic means of using HebbianLinks in inferential Atom-choice is simple: If there are Atoms linked via HebbianLinks with the other Atoms in the inference tree, then these Atoms should be given preference in the Evaluator's (bandit problem based) selection process.

Along the same lines but more subtly, another valuable heuristic for guiding inference control is "on-the-fly associatedness assessment." If there is a chance to apply the chosen Evaluator via working with Atoms that are

- strongly associated with the Atoms in the Atom being evaluated (via HebbianLinks)
- strongly associated with each other via HebbianLinks (hence forming a *cohesive set*)

then this should be ranked as a good thing.

For instance, it may be the case that, when doing deduction regarding relationships between humans, using relationships involving other humans as intermediate nodes in the deduction is often useful. Formally this means that, when doing inference of the form

```
Inheritance (Evaluation human A) B
Inheritance (Evaluation human C) B
|-
Inheritance A C
```

then it is often valuable to choose B so that

```
HebbianLink B human
```

has high strength. This would follow from the above-mentioned heuristic.

Next, suppose one has noticed a more particular rule — that in trying to reason about humans, it is particularly useful to think about their wants. This suggests that in abductions of the above form it is often useful to choose B of the form

```
B = SatisfyingSet [ wants( human $X, concept C) ]
```

This is too fine-grained a cognitive-control intuition to come from simple association-following. Instead, it requires fairly specific data-mining of the system's inference history. It requires the recognition of "Hebbian predicates" of the form

```
HebbianImplication[ ForAll $B]
  AND
    Inheritance $A human
    Inheritance $B human
    ThereExists $C
      Equivalence
        $B
        SatisfyingSet[$X]
          Evaluation wants ($X, $C)
  AND
    Inheritance $A $B
    Inheritance $C $B
```

The semantics of

```
HebbianImplication X Y
```

is that *when X is being thought about, it is often valuable to think about Y shortly thereafter.*

So what is required to do inference control according to heuristics like *think about humans according to their wants* is a kind of backward-chaining inference that combines Hebbian implications with PLN inference rules. PLN inference says that to assess the relationship between two people, one approach is abduction. But Hebbian learning says that when setting up an abduction between two people, one useful precondition is if the intermediate term in the abduction regards wants. Then a check can be made whether there are any relevant intermediate terms regarding wants in the system's memory.

What we see here is that the overall inference control strategy can be quite simple. For each Evaluator that can be applied, a check can be made for whether there is any relevant Hebbian knowledge regarding the general constructs involved in the Atoms this Evaluator would be manipulating. If so, then the prior probability of this Evaluator is increased, for the purposes of the Evaluator-choice bandit problem. Then, if the Evaluator is chosen, the specific Atoms this Evaluator would involve in the inference can be summoned up, and the relevant Hebbian knowledge regarding these Atoms can be utilized.

To take another similar example, suppose we want to evaluate

```
Inheritance pig dog
```

via the deduction Evaluator (which also carries out induction and abduction). There are a lot of possible intermediate terms, but a reasonable heuristic is to ask a few basic questions about them: How do they move around? What do they eat? How do they reproduce? How intelligent are they? Some of these standard questions correspond to particular intermediate terms, e.g. the intelligence question partly boils down to computing

```
Inheritance pig intelligent
```

and

```
Inheritance dog intelligent
```

So a link

```
HebbianImplication animal intelligent
```

may be all that's needed to guide inference to asking this question. This HebbianLink says that when thinking about animals, it's often interesting to think about intelligence. This should bias the deduction Evaluator to choose intelligent as an intermediate node for inference.

On the other hand, the *what do they eat* question is subtler and boils down to asking: Find $X so that when

```
R($X) = SatisfyingSet[$Y] eats ($Y,$X)
```

holds, then we have

```
Inheritance pig R($X)
```

and

```
Inheritance dog R($X)
```

In this case, a HebbianLink from animal to eat would not really be fine-grained enough. Instead we want a link of the form

```
HebbianImplication
    Inheritance $X animal
    SatisfyingSet[$Y] eats ($Y,$X)
```

This says that when thinking about an animal, it's interesting to think about what that animal eats.

The deduction Evaluator, when choosing which intermediate nodes to use, needs to look at the scope of available HebbianLinks and HebbianPredicates and use them to guide its choice. And if there are no good intermediate nodes available, it may report that it doesn't have enough experience to assess with any confidence whether it can come up with a good conclusion. As a consequence of the bandit-problem dynamics, it may be allocated reduced resources, or another Evaluator is chosen altogether.

Incorporating Other Cognitive Processes Into Inference

Hebbian inference control is a valuable and powerful process, but it is not always going to be enough. The solution of some problems that OCP chooses to address via inference will ultimately require the use of other methods, too. In these cases, one workaround is for inference to call on other cognitive processes to help it out.

This is done via the forward or backward chainer identifying specific Atoms deserving of attention by other cognitive processes, and then spawning Tasks executing these other cognitive processes on the appropriate Atoms.

Firstly, which Atoms should be selected for this kind of attention? What we want are InferenceTreeNodes that:

- have high STI

- have the impact to significantly change the overall truth value of the inference tree they are embedded in (something that can be calculated by hypothetically varying the truth value of the InferenceTreeNode and seeing how the truth value of the overall conclusion is affected)
- have truth values that are known with low confidence

Truth values meeting these criteria should be taken as strong candidates for attention by other cognitive processes.

The next question is which other cognitive processes do we apply in which cases?

MOSES in supervised categorization mode can be applied to a candidate InferenceTreeNode representing a OCP Node if it has a sufficient number of members (Atoms linked to it by MemberLinks); and, a sufficient number of new members have been added to it (or have had their membership degree significantly changed) since MOSES in supervised categorization mode was used on it last.

Next, pattern mining can be applied to look for connectivity patterns elsewhere in the AtomTable, similar to the connectivity patterns of the candidate Atom, if the candidate Atom has changed significantly since pattern mining last visited it

More subtly, what if for example we try to find whether "Swedishness" implies "Ugliness", and we know that "bad genes" implies Ugliness, but can't find a way, by backward chaining, to prove that Swedishness implies "bad genes". Then we could launch a non-backward-chaining algorithm to measure the overlap of SatisfyingSet(Swedishness) and SatisfyingSet(bad genes). Specifically, we could use MOSES in supervised categorization mode to find relationships characterizing Swedishness and other relationships characterizing "bad genes", and then do some forward chaining inference on these relationships. This would be a general heuristic for what to do when there's a link with low confidence but high potential importance to the inference tree.

Contextually Adaptive Confidence Decay

PLN is all about uncertain truth values, yet in itself, there is an important kind of uncertainty it doesn't handle explicitly and completely in its standard truth value representations: the decay of information with time.

PLN does have an elegant mechanism for handling this: in the <s,d> formalism for truth values, strength s may remain untouched by time (except as new evidence specifically corrects it), but d may decay over time. So, our confidence in our old observations decreases with time. In the indefinite probability formalism, what this means is that old truth value intervals get wider, but retain the same mean as they had back in the good old days.

But the tricky question is: How fast does this decay happen?

This can be highly context-dependent.

For instance, 20 years ago I learned that the electric guitar is the most popular instrument in the world, and also that there are more bacteria than humans on Earth. The former fact is no longer true (keyboard synthesizers have outpaced electric guitars), but the latter is. And, if you'd asked me 20 years ago which fact would be more likely to become obsolete, I would have answered the former — because I knew particulars of technology would likely change far faster than basic facts of biology.

It seems to me that estimating confidence decay rates for different sorts of knowledge in different contexts is a tractable data mining problem, that can be solved via the system keeping a record of the observed truth values of a random sampling of Atoms as they change over time. (Operationally, this record may be maintained in parallel with the SystemActivityTable and other tables maintained for purposes of effort estimation, attention allocation and credit assignment.) If the truth values of a certain sort of Atom in a certain context change a lot, then the confidence decay rate for Atoms of that sort should be increased.

This can be quantified nicely using the indefinite probabilities framework.

For instance, we can calculate, for a given sort of Atom in a given context, separate b-level credible intervals for the L and U components of the Atom's truth value at time t-r, centered about the corresponding values at time t. (This would be computed by averaging over all t values in the relevant past, where the relevant past is defined as some particular multiple of r; and over a number of Atoms of the same sort in the same context.)

Since historically-estimated credible-intervals won't be available for every exact value of r, interpolation will have to be used between the values calculated for specific values of r.

Also, while separate intervals for L and U would be kept for maximum accuracy, for reasons of pragmatic memory efficiency one might want to maintain only a single number x, considered as the radius of the confidence interval about both L and U. This could be obtained by averaging together the empirically obtained intervals for L and U.

Then, when updating an Atom's truth value based on a new observation, one performs a revision of the old TV with the new, but before doing so, one first ***widens*** the interval for the old one by the amounts indicated by the above-mentioned credible intervals.

For instance, if one gets a new observation about A (L_new, U_new), and the prior TV of A (L_old, U_old) is 2 weeks old, then one may calculate that L_old should really be considered as

```
(L_old - x, L_old+x)
```

and U_old should really be considered as

```
(U_old - x, U_old + x)
```

so that (L_new, U_new) should actually be revised with

```
(L_old - x, U_old + x)
```

to get the total

```
(L,U)
```

for the Atom after the new observation.

Note that we have referred fuzzily to "sort of Atom" rather than "type of Atom" in the above. This is because Atom type is not really the right level of specificity to be looking at. Rather — as in the guitar vs. bacteria

example above — confidence decay rates may depend on semantic categories, not just syntactic (Atom type) categories. To give another example, confidence in the location of a person should decay more quickly than confidence in the location of a building. So ultimately confidence decay needs to be managed by a pool of learned predicates, which are applied periodically. These predicates are mainly to be learned by data mining, but inference may also play a role in some cases.

The ConfidenceDecay MindAgent must take care of applying the confidence-decaying predicates to the Atoms in the AtomTable, periodically.

The ConfidenceDecayUpdater MindAgent must take care of:

- forming new confidence-decaying predicates via data mining, and then revising them with the existing relevant confidence-decaying predicates
- flagging confidence-decaying predicates which pertain to important Atoms but are unconfident, by giving them STICurrency, so as to make it likely that they will be visited by inference

An Example

As an example of the above issues, consider that the confidence decay of

```
Inh Ari male
```

should be low whereas that of

```
Inh Ari tired
```

should be higher, because we know that for humans, being male tends to be a more permanent condition than being tired.

This suggests that concepts should have context-dependent decay rates, e.g. in the context of humans, the default decay rate of maleness is low whereas the default decay rate of tired-ness is high.

However, these defaults can be overridden. For instance, one can say "As he passed through his 80's, Grandpa just got tired, and eventually he died." This kind of tiredness, even in the context of humans, does not have a rapid decay rate. This example indicates why the confidence decay rate of a particular Atom needs to be able to override the default.

Implementationally, one mechanism to achieve the above example would be as follows. One could incorporate an *interval* confidence decay rate as an optional component of a truth value. As noted above one can keep two separate intervals for the L and U bounds; or to simplify things one can keep a single interval and apply it to both bounds separately.

Then, e.g., to define the decay rate for tiredness among humans, we could say:

```
ImplicationLink_HOJ
   InheritanceLink $X human
   InheritanceLink $X tired <confidenceDecay = [0,.1]>
```

or else (preferably)

```
ContextLink
  human
  InheritanceLink $X tired <confidenceDecay = [0,.1]>
```

Similarly, regarding maleness we could say

```
ContextLink
  human
  Inh $X male <confidenceDecay = [0,.00001]>
```

Then one way to express the violation of the default in the case of grandpa's tiredness would be:

```
InheritanceLink grandpa tired <confidenceDecay = [0,.001]>
```

(Another way to handle the violation from default, of course, would be to create a separate Atom

```
tired_from_old_age
```

and consider this as a separate sense of "tired" from the normal one, with its own confidence decay setting.)

In this example we see that, when a new Atom is created (e.g. InheritanceLink Ari tired), it needs to be assigned a confidence decay rate via inference based on relations such as the ones given above (this might be done e.g. by placing it on the queue for immediate attention by the ConfidenceDecayUpdater MindAgent). And periodically its confidence decay rate could be updated based on ongoing inferences (in case relevant abstract knowledge about confidence decay rates changes). Making this sort of inference reasonably efficient might require creating a special index containing abstract relationships that tell you something about confidence decay adjustment, such as the examples given above.

Chapter Seven: Integrative Inference

Integrative Inference

Adaptive Inference Control

In the previous subsection we have seen an example of what the PLN inference rules can do when applied to a small test database. They can draw plausible conclusions based on the knowledge at hand. But of course, inference on a small test database is quite different from real-world inference. There is little doubt that in principle PLN is capable of complex real-world inference, in the sense that there exist meaningful chains of PLN inferences leading from the data at an embodied AGI system's disposal to the conclusions the system needs to draw to achieve its complex goals. The real question, however, is how the meaningful inference-chains are found from amidst the vast mass of meaningless ones. This is the essence of the question of *inference control.*

It is clear that in humans, inference control is all about context. We use different inference strategies in different contexts, and learning these strategies is most of what *learning to think* is all about. One might think to

approach this aspect of cognition, in the OCP design, by introducing a variety of different inference control heuristics, each one giving a certain algorithm for choosing which inferences to carry out in which order in a certain context. However, in keeping with the *integrated intelligence* theme that pervades OCP, we have chosen an alternate strategy for PLN. We have one inference control scheme, which is quite simple, but which relies partially on structures coming from outside PLN proper. The requisite variety of inference control strategies is provided by variety in the non-PLN structures such as

- OpenCogPrime:HebbianLinks existing in the AtomTable.
- Patterns recognized via pattern-mining in the corpus of prior inference trails

[edit] Mechanics of the Inference Control Process

We will now describe the basic "inference control" loop of PTL in OCP. We will discuss it in the context of backward chaining; the case of forward chaining is very similar.

Given an expression to evaluate via inference (according to any one of the query operations mentioned above), there is a collection of Evaluator objects that matches the expression.

First, each Evaluator object makes a preliminary assessment regarding how likely it is to come up with decent results. This assessment is made based on some preliminary explorations of what Atoms it might use to draw its conclusions — and study of various links (including HebbianLinks) that exist relating to its actions on these Atoms (we will give some examples of this shortly), and information regarding what Evaluators have been useful in what prior inferences in related contexts (stored in a structure called the InferencePatternRepository, to be discussed below).

Then, the overall evaluation process looks at the assessments made by each Evaluator and decides how much computational resource to allocate to each Evaluator. This we may call the "Evaluator choice problem" of inference control.

Finally, each Evaluator chosen, then needs to make choices regarding which Atoms to use in its inference — and this choice must be made by use of existing links, and information in the InferencePatternRepository. This is the "Atom choice problem" of inference control.

As an example of the choices to be made by an individual Evaluator, consider that to evaluate (Inheritance A C) via a deduction-based Evaluator, some collection of intermediate nodes for the deduction must be chosen. In the case of higher-order deduction, each deduction may involve a number of complicated subsidiary steps, so perhaps only a single intermediate node will be chosen. This choice of intermediate nodes must also be made via context-dependent probabilities. In the case of other Evaluators besides deduction, other similar choices must be made.

So the basic inference algorithm we have discussed is basic backward-chaining inference, but aggressive and adaptive pruning using HebbianLinks and other existing knowledge is done at every stage.

The Evaluator Choice Problem as a Bandit Problem

The evaluator choice problem, as described above, is an example of a "multi-armed bandit problem" as commonly studied in statistics. The atom choice problem is also a bandit problem. Both of these problems can be approached via using standard "bandit problem" heuristics.

The paradigm bandit problem involves a slot machine ("multi-armed bandit") with N levers to pull, each of which may have a different odds of yielding a reward each time it is pulled. The player's goal is to maximize her earnings, but when she starts out playing with the bandit, she has no idea which levers lead to which levels of reward. So, at any given point, she must balance two factors:

- Exploitation: pulling the level that seems to give maximum reward, based on experience so far
- Exploration: trying out various levers to get a sample of their performance, so as to get more confident estimates of the reward level offered by each lever

Obviously, as more experience is gained, the bias should shift from exploration towards exploitation. There is a substantial body of mathematics regarding bandit problems, but most of the results prove inapplicable in real-world situations due to making inappropriate statistical assumptions. In practice, the two most common algorithms for solving bandit problems are:

- epsilon-greedy: spend 1-epsilon of your time exploiting the best option found so far (with "best" defined in terms of expected reward), and epsilon of your time randomly exploring other options
- softmax: assign a probability to each option, using a heuristics formula based on thermodynamics that assigns higher probabilities to options that have proved more successful in the past, with an exponential scaling that favors successful options nonlinearly over unsuccessful ones

The only modification we choose to make to these simple algorithms in the OCP context is to replace the probability with a product

```
probability * weight_of_evidence
```

This evidence-weighted probability may be used within both epsilon-greedy and softmax.

Choosing an Evaluator for an inference step within PLN is a bandit problem, where prior experience is used to provide initial probabilities for the options (which are possible Evaluators rather than levers to pull), and then inferences done using each option are used to provide increasingly confident probabilities for each option. The default approach within OCP is softmax, though epsilon-greedy is also provided and may prove better in some contexts.

In Atom selection, the options (levers) are Atoms to use within an inference rule (an Evaluator).

It is important to note, however, that the statistical nature of the Atom-choice bandit problem is different from the statistics of the Evaluator-choice bandit problem, because there are not that many Evaluators, but there are a lot of Atoms. It would be possible to merge the two into a single bandit problem, whose options were (Evaluator, Atom-set) tuples, but this seems an inferior heuristic approach. The Evaluator bandit problem involves a relatively small number of options, which makes it more tractable. So we have chosen to break down the inference control process into two levels of bandit problems: Evaluator choice and Atom choice.

The complexity of the inference control problem can be well-understood by observing that each individual step poses two difficult statistical inference problems, one nested within the other! What allows pragmatic inferences to be achievable at all is, clearly, prior knowledge, which allows most of the bandit problems occurring in a complex inference to be "pre-solved" via assumption of prior probabilities. Normally, only a handful of difficult steps in an inference need to actually be studied statistically via numerous iterations of the epsilon-greedy or softmax algorithms. On the other hand, the first few inferences in an unfamiliar domain may not connect with the system's knowledge base of prior probabilities, and may thus need to be done in a much slower, more statistically thorough way.

Inference Pattern Mining

Among the data used to guide the solution of the Evaluator choice problem, the most important component is explicit information regarding which Evaluators have been useful in which contexts during past inferences.

This information is stored in OCP in a data repository called the InferencePatternRepository — which is, quite simply, a special "data table" containing inference trees and patterns recognized therein. An "inference tree" refers to a tree whose nodes are Atoms (generally: Atom-versions), and whose links are inference steps (so each link is labeled with a certain Evaluator).

Note that, in many cases, PLN creates a variety of exploratory inference trees internally, in the context of doing a single inference. Most of these inference trees will never be stored in the AtomTable, because they are unconfident and may not have produced extremely useful results. However, they should still be stored in the InferencePatternRepository. Ideally one would store all inference trees there. In a large OCP system this may not be feasible, but then a wide variety of trees should still be retained, including mainly successful ones but also a sampling of unsuccessful ones for purpose of comparison.

The InferencePatternRepository may then be used in two ways:

- An inference tree being actively expanded (i.e. utilized within the PLN inference system) may be compared to inference trees in the repository, in real time, for guidance. That is, if a node N in an inference tree is being expanded, then the repository can be searched for nodes similar to N, whose contexts (within their inference trees) are similar to the context of N within its inference tree. A study can then be made regarding which Evaluators and Atoms were most useful in these prior, similar inferences, and the results of this can be used to guide ongoing inference.
- Patterns can be extracted from the store of inference trees in the InferencePatternRepository, and stored separately from the actual inference trees (in essence, these patterns are inference subtrees with variables in place of some of their concrete nodes or links). An inference tree being expanded can then be compared to these patterns instead of, or in addition to, the actual trees in the repository. This provides greater efficiency in the case of common patterns among inference trees.

A reasonable approach may be to first check for inference patterns and see if there are any close matches; and if there are not, to then search for individual inference trees that are close matches.

Mining patterns from the repository of inference trees is a potentially highly computationally expensive operation, but this doesn't particularly matter since it can be run periodically in the background while inference proceeds at its own pace in the foreground, using the mined patterns. Algorithmically, it may be done either by

exhaustive frequent-itemset-mining (as in the Apriori or relim algorithms), or by stochastic greedy mining. These operations should be carried out by an InferencePatternMiner.

Hebbian Inference Control

One aspect of inference control is Evaluator choice, which is based on mining contextually relevant information from the InferencePatternRepository (see OpenCogPrime:InferencePatternMining). But, what about the Atom choice aspect? This can in some cases be handled via the OpenCogPrime:InferencePatternRepository as well, but it is less likely to serve the purpose than in the case of Evaluator choice. Evaluator choice is about finding structurally similar inferences in roughly similar contexts, and using them as guidance. But Atom choice has a different aspect: it is also about what Atoms have tended to be related to the other Atoms involved in an inference, generically, not just in the context of prior inferences, but in the context of prior perceptions, cognitions and actions in general.

Concretely, this means that Atom choice must make heavy use of HebbianLinks. The formation of HebbianLinks will be discussed in the following chapter, on attention allocation. Here it will suffice to get across the basic idea. The discussion of HebbianLinks here will hopefully serve to help you understand the motivation for the HebbianLink formation algorithm to be discussed later. Of course, inference is not the only user of HebbianLinks in the OCP system, but its use of HebbianLinks is somewhat representative.

The semantics of a HebbianLink between A and B is, intuitively: In the past, when A was important, B was also important. HebbianLinks are created via two basic mechanisms: pattern-mining of associations between importances in the system's history, and PLN inference based on HebbianLinks created via pattern mining (and inference) Thus, saying that PLN inference control relies largely on HebbianLinks is in part saying that PLN inference control relies on PLN. There is a bit of a recursion here, but it's not a bottomless recursion because it bottoms out with HebbianLinks learned via pattern mining.

As an example of the Atom-choices to be made by an individual Evaluator in the course of doing inference, consider that to evaluate (Inheritance A C) via a deduction-based Evaluator, some collection of intermediate nodes for the deduction must be chosen. In the case of higher-order deduction, each deduction may involve a number of complicated subsidiary steps, so perhaps only a single intermediate node will be chosen. This choice of intermediate nodes must be made via context-dependent prior probabilities. In the case of other Evaluators besides deduction, other similar choices must be made.

The basic means of using HebbianLinks in inferential Atom-choice is simple: If there are Atoms linked via HebbianLinks with the other Atoms in the inference tree, then these Atoms should be given preference in the Evaluator's (bandit problem based) selection process.

Along the same lines but more subtly, another valuable heuristic for guiding inference control is "on-the-fly associatedness assessment." If there is a chance to apply the chosen Evaluator via working with Atoms that are

- strongly associated with the Atoms in the Atom being evaluated (via HebbianLinks)
- strongly associated with each other via HebbianLinks (hence forming a *cohesive set*)

then this should be ranked as a good thing.

For instance, it may be the case that, when doing deduction regarding relationships between humans, using relationships involving other humans as intermediate nodes in the deduction is often useful. Formally this means that, when doing inference of the form

```
Inheritance (Evaluation human A) B
Inheritance (Evaluation human C) B
|-
Inheritance A C
```

then it is often valuable to choose B so that

```
HebbianLink B human
```

has high strength. This would follow from the above-mentioned heuristic.

Next, suppose one has noticed a more particular rule — that in trying to reason about humans, it is particularly useful to think about their wants. This suggests that in abductions of the above form it is often useful to choose B of the form

```
B = SatisfyingSet [ wants( human $X, concept C) ]
```

This is too fine-grained a cognitive-control intuition to come from simple association-following. Instead, it requires fairly specific data-mining of the system's inference history. It requires the recognition of "Hebbian predicates" of the form

```
HebbianImplication[ ForAll $B]
  AND
    Inheritance $A human
    Inheritance $B human
    ThereExists $C
      Equivalence
        $B
        SatisfyingSet[$X]
          Evaluation wants ($X, $C)
  AND
    Inheritance $A $B
    Inheritance $C $B
```

The semantics of

```
HebbianImplication X Y
```

is that *when X is being thought about, it is often valuable to think about Y shortly thereafter.*

So what is required to do inference control according to heuristics like *think about humans according to their wants* is a kind of backward-chaining inference that combines Hebbian implications with PLN inference rules. PLN inference says that to assess the relationship between two people, one approach is abduction. But Hebbian learning says that when setting up an abduction between two people, one useful precondition is if the intermediate term in the abduction regards wants. Then a check can be made whether there are any relevant intermediate terms regarding wants in the system's memory.

What we see here is that the overall inference control strategy can be quite simple. For each Evaluator that can be applied, a check can be made for whether there is any relevant Hebbian knowledge regarding the general constructs involved in the Atoms this Evaluator would be manipulating. If so, then the prior probability of this Evaluator is increased, for the purposes of the Evaluator-choice bandit problem. Then, if the Evaluator is chosen, the specific Atoms this Evaluator would involve in the inference can be summoned up, and the relevant Hebbian knowledge regarding these Atoms can be utilized.

To take another similar example, suppose we want to evaluate

```
Inheritance pig dog
```

via the deduction Evaluator (which also carries out induction and abduction). There are a lot of possible intermediate terms, but a reasonable heuristic is to ask a few basic questions about them: How do they move around? What do they eat? How do they reproduce? How intelligent are they? Some of these standard questions correspond to particular intermediate terms, e.g. the intelligence question partly boils down to computing

```
Inheritance pig intelligent
```

and

```
Inheritance dog intelligent
```

So a link

```
HebbianImplication animal intelligent
```

may be all that's needed to guide inference to asking this question. This HebbianLink says that when thinking about animals, it's often interesting to think about intelligence. This should bias the deduction Evaluator to choose intelligent as an intermediate node for inference.

On the other hand, the *what do they eat* question is subtler and boils down to asking: Find $X so that when

```
R($X) = SatisfyingSet[$Y] eats ($Y,$X)
```

holds, then we have

```
Inheritance pig R($X)
```

and

```
Inheritance dog R($X)
```

In this case, a HebbianLink from animal to eat would not really be fine-grained enough. Instead we want a link of the form

```
HebbianImplication
    Inheritance $X animal
    SatisfyingSet[$Y] eats ($Y,$X)
```

This says that when thinking about an animal, it's interesting to think about what that animal eats.

The deduction Evaluator, when choosing which intermediate nodes to use, needs to look at the scope of available HebbianLinks and HebbianPredicates and use them to guide its choice. And if there are no good intermediate nodes available, it may report that it doesn't have enough experience to assess with any confidence whether it can come up with a good conclusion. As a consequence of the bandit-problem dynamics, it may be allocated reduced resources, or another Evaluator is chosen altogether.

Incorporating Other Cognitive Processes Into Inference

Hebbian inference control is a valuable and powerful process, but it is not always going to be enough. The solution of some problems that OCP chooses to address via inference will ultimately require the use of other methods, too. In these cases, one workaround is for inference to call on other cognitive processes to help it out.

This is done via the forward or backward chainer identifying specific Atoms deserving of attention by other cognitive processes, and then spawning Tasks executing these other cognitive processes on the appropriate Atoms.

Firstly, which Atoms should be selected for this kind of attention? What we want are InferenceTreeNodes that:

- have high STI
- have the impact to significantly change the overall truth value of the inference tree they are embedded in (something that can be calculated by hypothetically varying the truth value of the InferenceTreeNode and seeing how the truth value of the overall conclusion is affected)
- have truth values that are known with low confidence

Truth values meeting these criteria should be taken as strong candidates for attention by other cognitive processes.

The next question is which other cognitive processes do we apply in which cases?

MOSES in supervised categorization mode can be applied to a candidate InferenceTreeNode representing a OCP Node if it has a sufficient number of members (Atoms linked to it by MemberLinks); and, a sufficient number of new members have been added to it (or have had their membership degree significantly changed) since MOSES in supervised categorization mode was used on it last.

Next, pattern mining can be applied to look for connectivity patterns elsewhere in the AtomTable, similar to the connectivity patterns of the candidate Atom, if the candidate Atom has changed significantly since pattern mining last visited it

More subtly, what if for example we try to find whether "Swedishness" implies "Ugliness", and we know that "bad genes" implies Ugliness, but can't find a way, by backward chaining, to prove that Swedishness implies "bad genes". Then we could launch a non-backward-chaining algorithm to measure the overlap of SatisfyingSet(Swedishness) and SatisfyingSet(bad genes). Specifically, we could use MOSES in supervised categorization mode to find relationships characterizing Swedishness and other relationships characterizing "bad genes", and then do some forward chaining inference on these relationships. This would be a general

heuristic for what to do when there's a link with low confidence but high potential importance to the inference tree.

Contextually Adaptive Confidence Decay

PLN is all about uncertain truth values, yet in itself, there is an important kind of uncertainty it doesn't handle explicitly and completely in its standard truth value representations: the decay of information with time.

PLN does have an elegant mechanism for handling this: in the <s,d> formalism for truth values, strength s may remain untouched by time (except as new evidence specifically corrects it), but d may decay over time. So, our confidence in our old observations decreases with time. In the indefinite probability formalism, what this means is that old truth value intervals get wider, but retain the same mean as they had back in the good old days.

But the tricky question is: How fast does this decay happen?

This can be highly context-dependent.

For instance, 20 years ago I learned that the electric guitar is the most popular instrument in the world, and also that there are more bacteria than humans on Earth. The former fact is no longer true (keyboard synthesizers have outpaced electric guitars), but the latter is. And, if you'd asked me 20 years ago which fact would be more likely to become obsolete, I would have answered the former — because I knew particulars of technology would likely change far faster than basic facts of biology.

It seems to me that estimating confidence decay rates for different sorts of knowledge in different contexts is a tractable data mining problem, that can be solved via the system keeping a record of the observed truth values of a random sampling of Atoms as they change over time. (Operationally, this record may be maintained in parallel with the SystemActivityTable and other tables maintained for purposes of effort estimation, attention allocation and credit assignment.) If the truth values of a certain sort of Atom in a certain context change a lot, then the confidence decay rate for Atoms of that sort should be increased.

This can be quantified nicely using the indefinite probabilities framework.

For instance, we can calculate, for a given sort of Atom in a given context, separate b-level credible intervals for the L and U components of the Atom's truth value at time t-r, centered about the corresponding values at time t. (This would be computed by averaging over all t values in the relevant past, where the relevant past is defined as some particular multiple of r; and over a number of Atoms of the same sort in the same context.)

Since historically-estimated credible-intervals won't be available for every exact value of r, interpolation will have to be used between the values calculated for specific values of r.

Also, while separate intervals for L and U would be kept for maximum accuracy, for reasons of pragmatic memory efficiency one might want to maintain only a single number x, considered as the radius of the confidence interval about both L and U. This could be obtained by averaging together the empirically obtained intervals for L and U.

Then, when updating an Atom's truth value based on a new observation, one performs a revision of the old TV with the new, but before doing so, one first ***widens*** the interval for the old one by the amounts indicated by the above-mentioned credible intervals.

For instance, if one gets a new observation about A (L_new, U_new), and the prior TV of A (L_old, U_old) is 2 weeks old, then one may calculate that L_old should really be considered as

```
(L_old - x, L_old+x)
```

and U_old should really be considered as

```
(U_old - x, U_old + x)
```

so that (L_new, U_new) should actually be revised with

```
(L_old - x, U_old + x)
```

to get the total

```
(L,U)
```

for the Atom after the new observation.

Note that we have referred fuzzily to "sort of Atom" rather than "type of Atom" in the above. This is because Atom type is not really the right level of specificity to be looking at. Rather — as in the guitar vs. bacteria example above — confidence decay rates may depend on semantic categories, not just syntactic (Atom type) categories. To give another example, confidence in the location of a person should decay more quickly than confidence in the location of a building. So ultimately confidence decay needs to be managed by a pool of learned predicates, which are applied periodically. These predicates are mainly to be learned by data mining, but inference may also play a role in some cases.

The ConfidenceDecay MindAgent must take care of applying the confidence-decaying predicates to the Atoms in the AtomTable, periodically.

The ConfidenceDecayUpdater MindAgent must take care of:

- forming new confidence-decaying predicates via data mining, and then revising them with the existing relevant confidence-decaying predicates
- flagging confidence-decaying predicates which pertain to important Atoms but are unconfident, by giving them STICurrency, so as to make it likely that they will be visited by inference

An Example

As an example of the above issues, consider that the confidence decay of

```
Inh Ari male
```

should be low whereas that of

```
Inh Ari tired
```

should be higher, because we know that for humans, being male tends to be a more permanent condition than being tired.

This suggests that concepts should have context-dependent decay rates, e.g. in the context of humans, the default decay rate of maleness is low whereas the default decay rate of tired-ness is high.

However, these defaults can be overridden. For instance, one can say "As he passed through his 80's, Grandpa just got tired, and eventually he died." This kind of tiredness, even in the context of humans, does not have a rapid decay rate. This example indicates why the confidence decay rate of a particular Atom needs to be able to override the default.

Implementationally, one mechanism to achieve the above example would be as follows. One could incorporate an *interval* confidence decay rate as an optional component of a truth value. As noted above one can keep two separate intervals for the L and U bounds; or to simplify things one can keep a single interval and apply it to both bounds separately.

Then, e.g., to define the decay rate for tiredness among humans, we could say:

```
ImplicationLink_HOJ
   InheritanceLink $X human
   InheritanceLink $X tired <confidenceDecay = [0,.1]>
```

or else (preferably)

```
ContextLink
  human
  InheritanceLink $X tired <confidenceDecay = [0,.1]>
```

Similarly, regarding maleness we could say

```
ContextLink
  human
  Inh $X male <confidenceDecay = [0,.00001]>
```

Then one way to express the violation of the default in the case of grandpa's tiredness would be:

```
InheritanceLink grandpa tired <confidenceDecay = [0,.001]>
```

(Another way to handle the violation from default, of course, would be to create a separate Atom

```
tired_from_old_age
```

and consider this as a separate sense of "tired" from the normal one, with its own confidence decay setting.)

In this example we see that, when a new Atom is created (e.g. InheritanceLink Ari tired), it needs to be assigned a confidence decay rate via inference based on relations such as the ones given above (this might be done e.g.

by placing it on the queue for immediate attention by the ConfidenceDecayUpdater MindAgent). And periodically its confidence decay rate could be updated based on ongoing inferences (in case relevant abstract knowledge about confidence decay rates changes). Making this sort of inference reasonably efficient might require creating a special index containing abstract relationships that tell you something about confidence decay adjustment, such as the examples given above.

The PLN inference framework is not discussed extensively in this wikibook because it has a whole other, paper book devoted to it, *Probabilistic Logic Networks*. However, that book is about PLN rather than OCP, and doesn't go into depth on the integration of PLN with OCP.

The purpose of this page and its children is to discuss issues specifically pertinent to the integration of PLN with other OCP processes. Some such issues are dealt with in other pages already, such as OpenCogPrime:EvolutionBasedInferenceControl which discusses PEL as a form of inference control. The issues not dealt with in other pages, are gathered here.

In the main the issues discussed here pertain to inference *control* rather than the nature of individual inference steps. This is because individual inference steps are the same in PLN whether PLN is being utilized within or outside of OCP. On the other hand, inference control is done within OCP in ways that wouldn't necessarily make sense in a standalone-PLN context. The main theme of the chapter is adaptive inference control, but this is only a viable option if there is some method besides PLN that is studying patterns among inferences, and/or studying inference-relevant patterns among Atoms. If one were doing complex PLN inference outside the context of an integrative AI framework containing appropriate non-inferential cognitive processes, one would need to take a quite different approach. In fact, our suggestion is that integrative AI is essentially the only workable approach to the control of higher-order inference in complex cases. Certainly, attempts to solve the problem with a narrow-AI approach have been made many times before and have not succeeded.

Types of PLN Query

The PLN implementation in OCP is complex and lends itself for utilization via many different methods. However, a convenient way to think about it is in terms of three basic backward-focused query operations:

- findtv, which takes in an expression and tries to find its the truth value
- findExamples, which takes an expression containing variables and tries to find concrete terms to fill in for the variables
- createExamples, which takes an expression containing variables and tries to create new Atoms to fill in for the variables, using *concept creation* heuristics as discussed in a later chapter, coupled with inference for evaluating the products of concept creation

and one forward-chaining operation:

- findConclusions, which takes a set of Atoms and seeks to draw the most interesting possible set of conclusions via combining them with each other and with other knowledge in the AtomTable

These inference operations may of course call themselves and each other recursively, thus creating lengthy chains of diverse inference.

(In an earlier version of PLN, these operations actually had the above names. I am not sure if these precise command names have survived into the current implementation, but the same functionality is there. Maybe this page should be updated.)

Findtv is quite straightforward, at the high level of discussion adopted here. Various inference rules may match the Atom; in our current PLN implementation, loosely described below, these inference rules are executed by objects called Evaluators. In the course of executing findtv, a decision must be made regarding how much attention to allocate to each one of these Evaluator objects, and some choices must be made by the objects themselves — issues that involve processes beyond pure inference, and will be discussed later in this chapter. Depending on the inference rules chosen, findtv may lead to the construction of inferences involving variable expressions, which may then be evaluated via findExamples or createExamples queries.

The findExamples operation, on the other hand, sometimes reduces to a simple search through the Atomspace, as in the test examples above. On the other hand, it can also be done in a subtler way. If the findExamples Evaluator wants to find examples of $X so that F($X), but can't find any, then its next step is to run another findExamples query, looking for $G so that

```
Implication $G F
```

and then running findExamples on G rather than F. But what if this findExamples query doesn't come up with anything? Then it needs to run a createExamples query on the same implication, trying to build a $G satisfying the implication.

Finally, forward-chaining inference (findConclusions) may be conceived of as a special heuristic for handling special kinds of findExample problems. Suppose we have K Atoms and want to find out what consequences logically ensue from these K Atoms, taken together. We can form the conjunction of the K Atoms (let's call it C), and then look for $D so that

```
Implication C $D
```

Conceptually, this can be approached via findExamples, which defaults to createExamples in cases where nothing is found. However, this sort of findExamples problem is special, involving appropriate heuristics for combining the conjuncts contained in the expression C, which embody the basic logic of forward-chaining rather than backward-chaining inference.

A Toy Example

To exemplify the above ideas, I will now give a detailed *log* of a PLN backward chainer as applied to resolve simple findtv and findExamples queries over a small test database, consisting of a few dozen relationships regarding a half-dozen people. The example shows the actual output from a prior version of the OCP/PLN testing framework that we used to qualitatively assess PLN performance, in mid-2005. The style of output from the current version differs (due to the greater generality of the current version): at the moment we no longer use the English-like output that was used in the prior version, because it become confusing in the context of highly complex examples.

In this example we begin with the findtv query

```
findtv friendOf(Amir,Osama)
```

The heuristic then searches in the knowledge base to find what relationships are known involving the terms in the query. It first finds the following relationship involving friendOf, which indicates the (probabilistic, not certain) symmetry of the friendOf relationship:

```
friendOf is symmetricRelation (0.65,0.72)
```

Note that symmetry is stored as a relationship like any other. This allows PTL to deal nicely with the cases of relationships that are only partially, probabilistically symmetric, such as friendOf and enemyOf.

The heuristic then searches for relations involving symmetricRelation, finding the following relationship, an Equivalence relationship that embodies the definition of symmetry

EQUIVALENCE TO IMPLICATION

```
if R000 is symmetricRelation then if and only if R000(X007,X008) then R000(X008,X007)
(0.995,0.985)
```

(Here the notation such as X008 refers to VariableNodes).

The system then applies this definition to the relationship friendOf, which involves a step of deduction:

VARIABLE INSTANTIATION

```
if friendOf is symmetricRelation then if and only if friendOf(X007,X008) then
friendOf(X008,X007) (0.995, 0.985)
```

DEDUCTION (*NODE TRUTH VALUE*)

```
if and only if friendOf(X007,X008) then friendOf(X008,X007) (0.654, 0.713)
```

Now it applies a step of variable instantiation, seeking to match the new link it has learned regarding friendOf with the other terms that were there in the original query.

VARIABLE INSTANTIATION

```
if and only if friendOf(Osama,Amir) then friendOf(Amir,Osama) (0.654,0.713)
```

This gives it a way to assess the truth value of friendOf(Amir, Osama). Namely: it now realizes that, since friendOf is symmetric, it suffices to evaluate friendOf(Osama, Amir). Thus it submits a findtv query of its own creation.

DEDUCTION (*NODE TRUTH VALUE*)

```
The truth value of friendOf(Amir, Osama) is unknown:
findtv friendOf(Osama, Amir)
```

```
no answer
```

But unfortunately, this query finds no answer in the system's knowledge base. The exploitation of the symmetry of friendOf was a dead end, and the inference heuristic must now backtrack and try something else. Going back to the start, it looks for another relationship involving the terms in the original query, and finds this one, which indicates that friendOf is transitive:

```
friendOf is transitiveRelation (0.4, 0.8)

if and only if R001 is transitiveRelation then if exists X010 such that
AND(R001(X009,X010),R000(X010,X011)) then R001(X009,X011) (0.99,0.99)
```

EQUIVALENCE TO IMPLICATION

```
if R001 is transitiveRelation then if exists X010 such that
AND(R001(X009,X010),R000(X010,X011)) then R001(X009,X011) (0.995,0.985)
```

VARIABLE INSTANTIATION

```
if friendOf is transitiveRelation then if exists X010 such that
AND(friendOf(Amir,X010),friendOf(X010,Osama)) then friendOf(Amir, Osama) (0.995,0.985)
```

DEDUCTION (*NODE TRUTH VALUE*)

```
if exists X010 such that AND(friendOf(Amir,X010),friendOf(X010,Osama))
then friendOf(Amir, Osama) (0.410, 0.808)
```

In this case, we end up with a findExamples query rather than a findtv query:

FIND INSTANCES

```
findExamples AND(friendOf(Amir,X010),friendOf(X010,Osama))

X010={Britney}

friendOf(Amir, Britney) (0.8, 0.3)

friendOf(Britney, Osama) (0.7, 0.6)
```

To resolve this find query, it must use the truth value formula for the AND operator

AND RULE

```
AND(friendOf(Amir, Britney),friendOf(Britney,Osama)) (0.56,0.3)
```

Since this evaluation yields a reasonably high truth value for the find query, the system decides it can plug in the variable assignment

```
X010={Britney}
```

Along the way it also evaluates

```
exists X010 such that AND(friendOf(Amir,X010),friendOf(X010,Osama)) (0.56,0.3)
```

And, more pertinently, it can use the transitivity of friendOf to assess the degree to which Amir is a friend of Osama:

DEDUCTION (*NODE TRUTH VALUE*)

```
friendOf(Amir, Osama) (0.238, 0.103)
```

There is nothing innovative or interesting about the inference trajectory followed here — the simple inference control heuristic is doing just what one would expect based on the knowledge and inference rules supplied to it. For dealing with a reasonably large-sized knowledge base, more sophisticated inference control heuristics are needed, because a simplistic strategy leads to combinatorial explosion.

What is important to observe here is how the probabilistic truth values are propagated naturally and sensibly through the chain of inferences. This requires the coordination of a number of different PLN inference rules with appropriately tuned parameters.

Because the knowledge base is so small, the revision rule wasn't invoked in the above example. It's very easy to see how it would come up in the same example with a slightly larger knowledge base, however. If there were more knowledge about the friends of Amir and Osama, then the system could carry out a number of inferences based on the symmetry and transitivity of friendOf, and revise the results together at the end.

PLN and Bayes Nets

Next, some comments on the relationship between PLN and Bayes Nets may be useful. We have not yet implemented such an approach, but it may well be that Bayes Nets methods can be a useful augmentation to PLN for certain sorts of inference (specifically, for inference on networks of knowledge that are relatively static in nature).

We can't use standard Bayes Nets as the primary way of structuring reasoning in OCP because OCP's knowledge network is loopy. The peculiarities that allow standard Bayes net belief propagation to work in standard loopy Bayes nets, don't hold up in OCP, because of the way you have to update probabilities when you're managing a very large network in interaction with a changing world, so that different parts of which get different amounts of focus. So in PLN we use different mechanisms (the "inference trail" mechanism) to avoid "repeated evidence counting" whereas in loopy Bayes nets they rely on the fact that in the standard loopy Bayes net configuration, extra evidence counting occurs in a fairly constant way across the network.

However, when you have within the AtomTable a set of interrelated knowledge items that you know are going to be static for a while, and you want to be able to query them probabilistically, then building a Bayes Net (i.e. "freezing" part of OCP's knowledge network and mapping it into a Bayes Net) may be useful. I.e., one way to accelerate some PLN inference would be:

1. Freeze a subnetwork of the AtomTable which is expected not to change a lot in the near future
2. Interpret this subnetwork as a loopy Bayes net, and use standard Bayesian belief propagation to calculate probabilities based on it

This would be a highly efficient form of "background inference" in certain contexts. (Note that this requires an "indefinite Bayes net" implementation that propagates indefinite probabilities through the standard Bayes-net local belief propagation algorithms, but this is not problematic.)

Chapter Eight: Attention Allocation and Credit Assignment

Attention Allocation

The critical factor shaping real-world general intelligence is resource constraint. Without this issue, we could just have simplistic program-space-search algorithms like AIXItl instead of complicated systems like the human brain or OCP. Resource constraint is managed implicitly within various components of OCP, for instance in the finite population size used in PEL algorithms, and the finite depth of forward or backward chaining inference trees in PLN. But there is also a component of OCP that manages resources on a global and cognitive-process-independent manner: the *attention allocation* component.

The general principles the attention allocation process should follow are easy enough to see: History should be used as a guide, and an intelligence should make probabilistic judgments based on its experience, guessing which resource-allocation decisions are likely to maximize its goal-achievement. The problem is that this is a difficult learning and inference problem, and to carry it out with excellent accuracy would require a limited-resources intelligent system to spend nearly all its resources deciding what to pay attention to and nearly none of them actually paying attention to anything else. Clearly this would be a very poor allocation of an AI system's attention! So simple heuristics are called for, to be supplemented by more advanced and expensive procedures on those occasions where time is available and correct decisions are particularly crucial.

In this page and its children

- OpenCogPrime:AttentionalDataMining
- OpenCogPrime:EconomicAttentionAllocation

We will describe how these *attention allocation* issues are addressed in the OCP design. Concretely, they are addressed via a set of mechanisms and equations for dynamically adjusting *importance values* attached to Atoms and MindAgents. Different importance values exist pertinent to different time scales, most critically the short-term (STI) and long-term (LTI) importances.

Two basic innovations are involved in the mechanisms attached to these importance values:

- treating attention allocation as a data mining problem: the system records information about what it's done in the past and what goals it's achieved in the past, and then recognizes patterns in this history and uses them to guide its future actions via probabilistically adjusting the (often context-specific) *importance values* associated with internal terms, actors and relationships, and adjusting the "effort estimates" associated with Tasks
- using an artificial-economics approach to update the importance values (attached to Atoms, MindAgents, and other actors in the OCP system) that regulate system attention

The integration of these two aspects is crucial. The artificial economics approach allows the system to make *rough and ready* attention allocation judgments in real time, whereas the data mining approach is slower and more resource-intensive but allows the system to make sophisticated attention allocation judgments when this is judged to be worth the effort.

In its particulars, this sort of thing is definitely not exactly what the human brain does, but we believe this is a case where slavish adherence to neuroscience is badly suboptimal. Doing attention allocation entirely in a distributed, formal-neural-nettish way is, we believe, extremely and unnecessarily inefficient, and given realistic resource constraints it necessarily leads to the rather poor attention allocation that we experience every day in our ordinary waking state of consciousness. Several aspects of attention allocation can be fruitfully done in a distributed, neural-nettish way, but not having a logically centralized repository of system-history information (regardless of whether it's physically distributed or not) simply can't be a good thing in terms of effective attention allocation. And we argue that, even for those aspects of attention allocation that are best addressed in terms of distributed, vaguely neural-nettish dynamics, an artificial-economics approach has significant advantages over a more strictly neural-net-like approach, due to the greater ease of integration with other cognitive mechanisms such as forgetting and data mining (more on this later).

A note on the role of inference in OCP attention allocation may be of value here. Although we are using probabilistic inference for attention allocation (along with other tools), the nature of this application of PLN is different from most other uses of PLN in OCP. Here — unlike in the *ordinary inference* case — we are not directly concerned with what conclusions the system draws, but rather, with what the system bothers to try drawing conclusions about. PLN, used in this context, effectively constitutes a nonlinear-dynamical iteration governing the flow of *attention* through the OCP system.

But, nevertheless, one should not underestimate the impact of attention allocation on the inferential conclusions that the OCP system draws. For example, if dynamics causes X to get such a low long-term-importance value that it's forgotten, then X will never get a chance to be used in reasoning again, which may lead the system to come to significantly different conclusions. Furthermore, importance updating and credit assignment guided schema execution guide the formation of new schemata and predicates, which are then incorporated into the system's declarative knowledge base by inferred higher-order links.

The structure of these wiki pages on attention allocation is as follows. After some brief conceptual comments, the semantics of STI and LTI values are discussed, with special attention to the role of LTI in the forgetting process. Then, the data mining approach to importance estimation is described, in terms of the definition of the SystemActivityTable that collects information about the OCP system's activities and its progress toward its goals. Finally, the economic approach to importance updating is described, including how it may incorporate information from the data mining approach when available.

Philosophy of Importance Updating

The "attention allocation" process — embodied in OCP in the process of dynamically adjusting "importance values" associated with Atoms, MindAgents and other OCP actors — is very simple: things that are important to the mind at any given time, should undergo more cognitive processing. And, things that are estimated likely to be important to the mind in the future, should be retained in memory.

But how do you tell what's important to the mind at a given time? Or what is likely to be important to the mind in the future? There are several aspects.

First, the external world can give relevant indications — something directly related to perception or action can demand attention.

Second, there is Peirce's Law of Mind — the heuristic that, if X is important, everything related to X should become a little more important. And, as a corollary, if several parts of a certain "holistic pattern" are important, then the other parts of the pattern should become important.

Finally, there is goal-driven importance. If something has to be thought about in order to achieve an important goal, then it should become important. Critical here is the perennial systemic goal of keeping the mind full of valuable relationships: if thinking about X has been valuable lately, then X should be a good candidate to be thought about again in the near future, unless other reasons override.

OCP's importance updating process provides explicit mechanisms that take all these aspects into account.

In the brain, of course, the importance updating process is achieved by a complex combination of chemicals acting in different parts of the brain in response to various signals. But neurophysiology and neuropsychology really give us very little specific guidance as to how this process works, except to suggest various general factors that are involved. We have designed our approach via a combination of introspective intuition, pragmatic simulation-running and hand-calculation, and practical experience with importance updating in OCP itself.

The closest analogue of the importance updating process in symbolic AI systems is the "blackboard system." Such a system contains an object called the "blackboard," onto which the most important entities in the system go. These systems are acted on in appropriate ways, the least important of them are removed from the blackboard, and new entities are placed onto the blackboard. What is unique about the OCP approach to importance updating is the way it integrates blackboard-system-style functionality with the nonlinear dynamics of huge population of agents.

Basic Atom Importance Measures

There are many kinds of patterns to be detected regarding the importance of attending various Atoms and sets of Atoms in various contexts. In some cases it is worth the system's while to study complex and subtle patterns of this form very closely. We will discuss the recognition of subtle attentional patterns later on . In many cases, though, only the simplest level of analysis can be done, for reasons of computational resource preservation.

Every Atom in OCP has a minimum amount of importance-analysis done on it, the result of which is contained in two numbers: the Short-Term Importance (STI) and the Long-Term Importance (LTI), which are contained in the AttentionValue objects associated with Atoms. STI and LTI are also associated with other OCP actors such as MindAgents, but for the moment we will discuss them only in the context of Atoms.

Roughly speaking, for Atoms, STI determines CPU allocation, in cases where a rough generic measure is needed. More concretely, many MindAgents use STI to make a default decision regarding which Atoms to pay attention to in the context of a particular cognitive process. Long-Term Importance (LTI), on the other hand, is

used to determine which nodes will be swapped out of RAM and onto disk, or deleted altogether. These numbers give a very coarse view of the value of attending the Atom in various circumstances, but they're far better than nothing; and collecting and calculating finer-grained information is often not practical for computational resource reasons.

The von Neumann architecture is obviously rearing its ugly head here: The split between Short-Term Importance and Long-Term Importance reflects the asymmetry between processor time and memory space in contemporary computers. Importance determines how much processor time an Atom should get; Long-Term Importance determines whether it should be allowed to remain in memory or not. The equations governing these quantities are tuned so that STI is allowed to change more quickly than LTI, because it's easy to de-allocate processor time to an Atom and then reallocate it again alter. LTI can't be as volatile, because once something is expelled from RAM and saved to disk, it's fairly computationally expensive to get it back. (In some cases it's very expensive to get it back — if one really wipes all direct trace of it from memory. On the other hand, if one keeps an index to an on-disk Atom without any other information about the Atom, then retaining it from disk is slow only because of disk access time rather than because of search effort.)

It can also be useful to consider further sorts of importance beyond STI and LTI. For instance, one may introduce VLTI (Very Long Term Importance), defined as the importance of saving an Atom on disk rather than deleting it permanently. VLTI should be even less volatile than LTI as permanent deletion is a very serious decision. However even disk space is not unlimited, and even more crucially, saving Atoms to disk consumes powerful processor time; so, for instance, it may make sense in some configurations for most perceptual data to have VLTI=0, so that when its LTI decays to 0 it is simply deleted rather than expending further effort by being saved. For most of the following we will discuss only STI and LTI; but VLTI may be handled identically to LTI, though with different parameter values, and this will be mentioned occasionally as appropriate.

How LTI and STI levels of Atoms are assigned and adjusted is a subtle issue to be discussed in later sections of this chapter.

Long-Term Importance and Forgetting

The LTI of an Atom indicates how valuable it is to the system to keep the Atom in question in memory. That is, it is roughly interpretable as the expected usefulness of having A in memory in the future. This has to do with the probability

```
P(A will be useful in the future)
```

The main point of calculating this number and keeping it around is to guide decisions regarding which Atoms to remove from memory (RAM) and which ones to retain.

How does the forgetting process (carried out by the Forgetting MindAgent) work? The first heuristic is to remove the Atoms with the lowest LTI, but this isn't the whole story. Clearly, the decision to remove an Atom from RAM should depend on factors beyond just the LTI of the Atom. For example, one should also take into account the expected difficulty in reconstituting A from other Atoms. Suppose the system has the relations:

dogs are animals

animals are cute

dogs are cute

and the strength of the third relation is not dissimilar from what would be obtained by deduction and revision from the first two relations and others in the system. Then, even if the system judges it will be very useful to know *dogs are cute* in the future, it may reasonably choose to remove *dogs are cute* from memory anyway, because it knows it can be so easily reconstituted. Thus, as well as removing the lowest-LTI Atoms, the Forgetting MindAgent should also remove Atoms meeting certain other criteria such as the combination of:

- low STI
- easy reconstitutability in terms of other Atoms that have LTI not less than its own

The goal of the "forgetting" process is to maximize the total utility of the Atoms in the Atomspace throughout the future.

Attention Allocation via Data Mining on the System Activity Table

In this section we'll discuss an object called the System Activity Table, which contains a number of subtables recording various activities carried out by the various objects in the OCP system. These tables may be used for sophisticated attention allocation processes, according to an approach in which importance values are calculated via direct data mining of a centralized knowledge store (the System Activity Table). This approach provides highly accurate attention allocation but at the cost of significant computational effort. In the following section we will discuss the logic and mechanics of its integration with less computationally expensive, less accurate techniques.

The System Activity Table

The System Activity Table is actually a set of tables, with multiple components.

First, there is a MindAgent Activity Table, which includes, for each MindAgent in the system, a table of the following form:

System Cycle	Effort Spent	Memory Used	Atom Combo 1 Utilized	Atom Combo 2 Utilized
Now	3.3	4000	Atom21, Atom44	Atom 44, Atom 47, Atom 345
Now -1	0.4	6079	Atom123, Atom33	Atom 345
...	...	...	...	...

The time-points recorded are the last T system cycles; the Atom-lists recorded are lists of Handles for Atoms.

The MindAgent's activity table records, for that MindAgent and for each system cycle, which Atom-sets were acted on by that MindAgent at that point in time.

Similarly, a table of this nature must be maintained for each Task-type, e.g. InferenceTask, MOSESCategorizationTask, etc. The Task tables are used to estimate Effort values for various Tasks, which are used in the procedure execution process. If it can be estimated how much spatial and temporal resources a Task is likely to use, via comparison to a record of previous similar tasks (in the Task table), then a MindAgent can decide whether it is appropriate to carry out this Task (versus some other one, or versus some simpler process not requiring a Task) at a given point in time, a process to be discussed in a later chapter.

In addition to the MindAgent and Task-type tables, we require that a table be maintained for each goal in the system, including simple goals like *create new truth value* as well as more complex and system-specific ones:

System Cycle	Total Achievement	Achievement for Atom	Achievement for set {Atom44, Atom 233}
Now — T	.4	.5	.4
Now —1	.5	.55	.4
...	...	...	...

For each goal, at minimum, the degree of achievement of the goal at a given time must be recorded. Optionally, at each point in time, the degree of achievement of a goal relative to some particular Atoms may be recorded. Typically the list of Atom-specific goal-achievements will be short and will be different for different goals and different time points. Some goals may be applied to specific Atoms or Atom sets, others may only be applied more generally.

Of course, if one has a large OCP configuration in which there are many functionally separate lobes, then separate tables must be used for each lobe.

The basic idea presented here is that attention allocation and credit assignment may be effectively carried out via datamining on these tables.

Semantics of Importance Levels Based on System Activity Tables

Now we return to our prior discussion of the semantics of STI and LTI, in the context of the specific data gathered in the SystemActivityTables.

Assume that the goals in the system are weighted, so that one can define a composite *ubergoal*, whose satisfaction is the weighted sum of the satisfactions of the goals in the system. (The proper definition of the ubergoal may also involve subtracting off for dependencies among the various goals.)

Let A denote an Atom, or else a set of more than one Atom. Where E denotes the expected value, we may define the generic STI of A as

```
I(A,T) = E(amount by which acting on A at time T will increase the satisfaction of the
ubergoal in the near future)
```

and the generic LTI of A as

```
LTI(A,T) = E(amount by which acting on A at time T will increase the satisfaction of the
ubergoal in the foreseeable future)
```

These may be estimated from the Activity Tables described above, using approximative approaches. The Activity Tables contain information on how much effort was applied to processing each Atom at each point in time, and how much utility was obtained from such application.

We may also define several more specific importance numbers, such as

```
I(A, M, G, R,T) = E( amount by which MindAgent M acting on A at time T will increase the
satisfaction of goal G by time T+R)
```

Generic importance may be defined in terms of specific importance by averaging specific importance over many argument values. It seems that a very useful kind of specific importance is

```
I(A, M,T) = E(amount by which MindAgent M acting on A at time T will increase the
satisfaction of the supergoal in the foreseeable future)
```

i.e., Mind-Agent-specific importance. This kind of importance can help MindAgents decide which Atoms to choose at a given time. Of course, if a given MindAgent lacks experience with a given Atom A, then I(A, M) may be estimated in terms of I(A), or in terms of I(B ,M) for other B that are related to A.

HebbianLinks

Next, special links called HebbianLinks may be created based on these importance values, recording probabilities such as

```
(HebbianLink A B).tv.s = E ( (A is important at time T).tv.s || (B is important at time
T).tv.s )
```

which should ideally be computed assuming the component truth values are distributional truth values, but can also be approximatively computed using simple truth values.

PLN inference may be carried out on HebbianLinks by treating (HebbianLink A B) as a virtual predicate evaluation relationship , i.e. as

```
EvaluationLink Hebbian_predicate (A, B)
```

PLN inference on HebbianLinks may then be used to update node importance values, because node importance values are essentially *node probabilities* corresponding to HebbianLinks. And similarly, MindAgent-relative node importance values are node probabilities corresponding to MindAgent-relative HebbianLinks.

Finally, inference on HebbianLinks leads to the emergence of maps, via the recognition of clusters in the graph of HebbianLinks.

Schema Credit Assignment

And, how do we apply a similar approach to clarifying the semantics of schema credit assignment?

From the above-described System Activity Tables, one can derive information of the form

```
Achieve(G,E,T) = "Goal G was achieved to extent E at time T"
```

which may be grounded as.

```
Similarity
    E
    ExOut
        GetTruthValue
        Evaluation
            atTime
                T
                HypLink G
```

and more refined versions such as

```
Achieve(G,E,T,A,P) = "Goal G was achieved to extent E using Atoms A (with parameters P)
at time T"

Enact(S,I,T1,O,T2) = "Schema S was enacted on inputs I at time T1, producing outputs O at
time T2"
```

The problem of schema credit assignment is then, in essence: Given a goal G and a distribution of times DT, figure out what schema to enact in order to cause G's achievement at some time in the future, where the desirability of times is weighted by DT.

The basic method used is the learning of predicates of the form

```
ImplicationLink
    F(C,P1,...,Pn)
    GS
```

where

- the Pi are Enact() statements in which the T1 and T2 are variable, and the S, I and O may be concrete or variable
- C is a predicate representing a *context*
- GS is an Achieve() statement, whose arguments may be concrete or abstract
- F is a Boolean function

Typically, the variable expressions in the T1 and T2 positions will be of the form T+offset, where offset is a constant value and T is a time value representing the time of inception of the whole compound schema. T may then be defined as TG - offset1, where offset1 is a constant value and TG is a variable denoting the time of achievement of the goal.

For the purpose of the present discussion, these predicates may be learned by any method; in practice, we assume they will be learned by a combination of greedy statistical pattern mining, PLN inference and PEL optimization.

The choice of what action to take at a given point in time is then a probabilistic decision. Based on the time-distribution DT given, the system will know a certain number of expressions CS = F(C,P1,...,Pn) of the type described above. Each of these will be involved in an ImplicationLink with a certain estimated strength. It may select the "compound schema" CS with the highest strength.

One might think to introduce other criteria here, e.g. to choose the schema with the highest strength but the lowest cost of execution. However, it seems better to include all pertinent criteria in the goal, so that if one wants to consider cost of execution, one assumes the existence of a goal that incorporates cost of execution (which may be measured in multiple ways, of course) as part of its internal evaluation function.

Another issue that arises is whether to execute multiple CS simultaneously. In many cases this won't be possible because two different CS's will contradict each other. It seems simplest to assume that CS's that can be fused together into a single plan of action, are presented to the schema execution process as a single fused CS. In other words, the fusion is done during the schema learning process rather than the execution process.

A question emerges regarding how this process deals with false causality, e.g. with a schema that, due to the existence of a common cause, often happens to occur immediately prior to the occurrence of a given goal. For instance, roosters crowing often occurs prior to the sun rising. The answer is: In the current approach, if roosters crowing often causes the sun to rise, then if the system wants to cause the sun to rise, it may well cause a rooster to crow. Once this fails, then the system will no longer hold the false believe, and afterwards will choose a different course of action. Furthermore, if it holds background knowledge indicating that roosters crowing is not likely to cause the sun to rise, then this background knowledge will be invoked by inference to discount the strength of the ImplicationLink pointing from rooster-crowing to sun-rising, so that the link will never be strong enough to guide schema execution in the first place.

The problem of credit assignment thus becomes a problem of creating appropriate heuristics to guide inference of ImplicationLinks of the form described above. Assignment of credit is then implicit in the calculation of truth values for these links. The difficulty is that the predicates F involved may be large and complex.

The Pragmatics of the Data Mining Approach to Attention Allocation

In the preceding sections we have discussed the probabilistic definition of STI, LTI and related quantities in terms of the SystemActivityTable. This is an elegant, conceptually and pragmatically correct approach to attention allocation; however, its Achilles Heel is computational efficiency. Data mining the SystemActivityTable is not necessarily as slow as doing sophisticated inference using PLN or learning complex new procedures or patterns using evolutionary learning — but even greedy stochastic pattern mining algorithms are not necessarily so cheap. Inference based on the SystemActivityTable can't be carried out regarding every Atom and MindAgent every second that a OCP system is operating.

What is needed is an attention allocation mechanism that incorporates these sophisticated methods together with cheaper and less intelligent methods, in a way that allows their seamless combination. This is achieved by the OpenCogPrime:EconomicAttentionAllocation methods described in the following section.

Economic Attention Allocation

We have discussed the semantics of STI, LTI and other measures of Atom importance, and have also described how the information collected in the SystemActivityTable may be data-mined to produce information pertinent to adjusting STI and LTI values. In this section we will build on these ideas and describe an approach to rapid, dynamic, *quick and dirty* attention allocation in OCP — which is able to incorporate information from the more sophisticated, more expensive data-mining approach when it is available.

In the economic approach, STI and LTI are represented as currencies. The amount of STI currently possessed by an entity (e.g. an Atom) is intended to be interpreted as roughly proportional to the STI as probabilistically defined above. However, a precise mapping from STI currency stores to probabilistic STI values does not ever need to be calculated in the course of practical attention allocation. Rather, estimations of probabilistic short-term importance may be used to affect STI currency values in other ways.

All "financial agents" within OCP own certain amounts of LTI and STI currency. Financial agents include at least: Units, MindAgents, and Atoms. Financial agents must exchange currencies among each other, according to equations designed to embody judgments of what allocation of currency is likely to optimize overall system effectiveness, and encourage overall proportionality between the currency store associated with an agent and the (not calculated in practice) probabilistically-defined STI and LTI values associated with the agent.

In the initial design for economic attention allocation within OCP, there is not an explicit competition among financial agents, in the sense that different agents are negotiating with each other or auctioning services to each other, each one seeking to obtain its own economic advantage. This sort of dynamic might be worth introducing into OCP later, but is not being suggested for initial exploration. Rather, the only "economic" thing about the currencies in the initial design is that they are conserved. If one agent's STI currency store is increased, another one's must be commensurately decreased. This conservation makes the overall dynamics of the system easier to understand and regulate, in a number of ways.

The "Attentional Focus" (AF) in this approach is simply defined as the set of Atoms with STI greater than a certain fixed threshold. This is important because some dynamics act differently on Atoms in AF than on other ones.

Within this basic framework, a number of kinds of currency exchange must occur. In the following I will qualitatively describe these, along with some other dynamics that are important for causing or regulating currency exchange.

Most of the discussion will pertain to generic STI and LTI values, however, the same exact mechanisms may be utilized for MindAgent-specific STI and LTI values, as regards Atoms. The only difference is that MindAgents do not get MindAgent-specific STI and LTI, so that for these values, the economics is entirely between the Atoms and the Lobe.

The basic logic of attentional currency flow to be described in this section is as follows. The flow occurs for STI and LTI separately:

- Each Unit is its own economy, there is no flow of currency between Units. Units contain *central banks,* and when a Unit's central bank carries out a financial transaction, we will say that the Unit carries out the transaction.
- Currency flows from Units to Atoms when a Unit rewards its component Atoms with *wages* for being utilized by MindAgents or by helping to achieve system goals
- Currency flows from Atoms to Units when Atoms pay Units rent for occupying the AtomTable
- Currency flows from Units to MindAgents when Units reward MindAgents for helping achieve system goals
- Currency flows from MindAgents to Units when MindAgents pay Units rent for the processor time they utilize
- Currency flows from Atoms to Atoms via *importance spreading* which utilizes HebbianLinks

The discussion here will mostly be qualitative. See the document AttentionAllocationEquations for specific, currently implemented equations along the lines discussed here.

(NOTE: This page may be obsolete in some of its details. Joel Pitt needs to update it to accord with the recent implementation of Attention Allocation he's done. Take it away, Joel!!!)

Atom Stimulation

In the economic approach to attention allocation, when an Atom is used by a MindAgent, it should get some economic reward for this (both STI and LTI). The reward may be greater if the Atom has been extremely useful to the MindAgent, especially if it's been extremely useful in helping the MindAgent achieve some important system goal.

One way to implement this would be for MindAgents to give some of their currency directly to the Atoms that they used; but we have decided not to go with this approach initially, though it may be useful in future. It seems overcomplex to have MindAgents restricted in the number of Atoms they can use, based on their STI currency store. Instead, as noted below in the discussion of MindAgent scheduling, in the current approach a MindAgent's currency store controls its processor time utilization, which indirectly affects the number of Atoms it can use.)

Instead, in the initial approach, MindAgents give "stimuli" to Atoms when using them. Then, each core cycle, during the importance updating process (see below), Atoms are rewarded proportionately to the amount of stimulus they have received (out of the Unit's fund of currency).

Importantly, this is where estimates of utility made by data mining the SystemActivityTable may come into play. A MindAgent may give a certain default amount of stimulus to an Atom merely for being utilized, but, if data mining has revealed that the Atom actually was important for what the MindAgent did, then two things may happen:

- The Unit may disburse a significantly greater amount of stimulus to the Atom
- The MindAgent-specific importances of the Atom may be incremented, via sending out MindAgent-specific stimuli

Atom Importance Updating

Next, the ImportanceUpdating MindAgent loops through all Atoms in the AtomTable and updates their STI and LTI values according to two principles:

- rewarding of Atoms (aka paying *wages* to Atoms) for getting stimuli (which are delivered by MindAgents and also based on SystemActivityTable data-mining, as mentioned above)
- charging of "rent" to Atoms

LTI rent payment occurs for all Atoms every time the ImportanceUpdating MindAgent carries out its loop through the AtomTable. On the other hand, STI rent payment occurs only for Atoms in the AttentionalFocus.

Initially, a constant tax rate has been assumed, although this assumption will be lifted in future versions. For instance, it makes sense for Atoms that consume a large amount of memory to be charged a differentially higher LTI rent.

The charging of rent has the effect of a *time decay* of Atom importance values, which is consistent with the probabilistic semantics of importance values: both STI and LTI are intended to measure *recent importance,* though STI works with a more stringent definition of recency. The rewarding of wages compensates for the time decay by adding new importance to the Atoms that have been found useful in recent past.

Homeostatic mechanisms

The dynamics of economic attention allocation is complex, and some special mechanisms are necessary to keep the dynamics of STI and LTI updating from becoming overly volatile:

- Adjustment of compensation or rental rates: The amount of currency given an Atom for each unit of stimulus, or the amount of rent charged to each Atom per cycle, may need to be adjusted periodically if the Atom STI or LTI values overall seem to be drifting systematically in some particular direction.
- Enforcement of caps; maximum and minimum values on the STI and LTI of any Atom are enforced. No one can get too rich or go into too much debt.

Hebbian Learning

Hebbian learning in OCP refers to the creation or truth-value-updating of HebbianLinks. As noted above, the general semantics of a HebbianLink from A to B is that when A is important, B is often also important. We may also have PredictiveHebbianLinks that have a time delay associated with them, similar to PredictiveImplicationLinks.

HebbianLink truth values may be updated by data mining on the SystemActivityTable; these updates will have fairly high weight of evidence, generally speaking. On the other hand, there are also lower-confidence heuristics for updating HebbianLink truth values. For instance, a simple heuristic is:

- if A and B are both in the AF at a certain point in time, then this is a piece of evidence in favor of the HL pointing from A to B.
- if at a certain point in time, A is in the AF but B is not, then this is a piece of evidence against the HL pointing from A to B

Of course this heuristic with its binary nature is somewhat crude, and more sensitive heuristics may easily be imagined.

What the SystemActivityTable data-mining will catch, and this simplistic approach will miss, are *Hebbian predicates* embodying criteria such as

```
"If both A and B are important at time T, and C is important at time T+1, then D is
likely to be important at time T+2"
```

Ultimately, PLN inference may be used to help determine the truth values of HebbianLinks and Hebbian predicates. This will allow the attention allocation process to be smarter than a simple reinforcement learning system.

Importance Spreading

Next, *importance spreading* refers to the phenomenon wherein an Atom decides to pass some of its currency to other Atoms to which it is related. This is in a sense a *poor man's inference* — it is preferable conceptually and ultimately more accurate to use PLN inference to infer one Atom's importance from the importance of other suitably related Atoms. But, computationally, doing this much inference is not always feasible. So, in the current design, the inference goes into the truth values of HebbianLinks — and HebbianLinks are then utilized within a non-inferential process of *importance spreading,* conducted by a separate ImportanceSpreading MindAgent.

For instance, if there is a Hebbian link from A to B, then A may want to pass some of its currency to B, depending on various factors including how much currency A has, and how many Hebbian links point out from A. Of course, the equations governing this must ensure that A doesn't pass too much of its currency to other Atoms, or A will have no currency left for itself. One approach is for each Atom to allocate a maximum percentage of its importance to spreading, and then spread this along its outgoing HebbianLinks proportionately.

The ultimate goal of this process is to heuristically approximate what would be achieved by using inference to estimate the implications of various Atoms' importance for various other Atoms' importance.

An interesting question is what happens, in OCP, when importance is spread to schemata or predicates? These schemata or predicates may spread some of their importance to their component schemata or predicates, but in order to guide this, it may be useful to have a specialized process for building HebbianLinks between Procedures and their component Procedures. The HebbianLink from procedure P to component procedure P1 should be given an STI proportional to the "importance" of P1 to P, which may be measured in various ways, e.g. by how much P changes if P1 is removed from P.

MindAgent Importance and Scheduling

So far we have discussed economic transactions between Atoms and Atoms, and between Atoms and Units. MindAgents have played an indirect role, via spreading stimulation to Atoms which causes them to get paid wages by the Unit. Now it is time to discuss the explicit role of MindAgents in economic transactions. This has to do with the integration of economic attention allocation with the Scheduler that schedules MindAgents.

This integration may be done in many ways, but one simple approach is:

1. When a MindAgent utilizes an Atom, this results in sending stimulus to that Atom. (Note that we don't want to make MindAgents pay for using Atoms individually; that would penalize MA's that use more Atoms, which doesn't really make much sense.)

2. MindAgents then get currency from the Lobe periodically, and get extra currency based on usefulness for goal achievement as determined by the credit assignment process. The Scheduler then gives more processor time to MindAgents with more STI.
3. However, any MindAgent with LTI above a certain minimum threshold will get some minimum amount of processor time (i.e. get scheduled at least once each N cycles).

As a final note: In a multi-Lobe Unit, the Unit may use the different LTI values of MA's in different Lobes to control the distribution of MA's among Lobes: e.g. a very important (LTI) MA might get cloned across multiple Lobes.

Chapter Nine: Embodiment: Perception and Action

Embodiment and its Importance for OpenCog Prime

Regarding the importance of embodiment for AI, expert opinions are all over the map.

1. Some AI theorists believe that robotic embodiment is necessary for the achievement of powerful AGI (Rodney Brooks for instance is a prominent champion of this perspective)
2. Others believe embodiment is entirely unnecessary, and one can create a human-level AGI exclusively using other mechanisms, e.g. textual conversation, interaction in formal logic, etc.
3. We believe embodiment is extremely convenient for AGI though perhaps not strictly necessary; and that virtual-world embodiment is an important, pragmatic and scalable approach to pursue alongside physical-robot embodiment. The essay http:/XXX on "Post-Embodied AI" elaborates this perspective in detail.

The natural synergy between advanced AI and gaming/virtual worlds has been avidly discussed for at least a decade (a milestone in this regard was John Laird's article XX), and is now finally becoming a practical reality. OpenCog will be integrated with a variety of virtual worlds including Multiverse and OpenSim.

Integrating Virtual Worlds with Robot Simulators

A critical question, for those who believe that embodiment is important for AGI, is: What must a world be, that an intelligent system may grow and learn within it?

And: What facilities must exist, within a mind, to mediate its interaction with other minds within this world?

Along these lines, it is arguable that current virtual world and gaming technology is not quite powerful and flexible enough to fully support virtually embodied AI. In the blog post XX it is argued that the way avatar actions are invoked in current virtual worlds is highly suboptimal from an AI perspective. Instead of having avatars controlled via a fixed set of animations programmed in programs like 3dsMax or specialized tools like those available for Second Life, avatars should have their joints controlled by simulated servomotors, as is done in robot simulators. Toward this end, integration of virtual world platforms with robotics simulation platforms such as Player/Gazebo would be one route.

Integration of OpenCog and Related Software with Virtual Worlds

A project to integrate OpenCog with the OpenSim virtual world is expected to start in late June 2008, and a Google Summer of Code student is also working on this during summer 2008.

Some related work has been done with the Novamente Cognition Engine:

1. a prototype integration of the NCE with AGISim, an experimental virtual world based on the CrystalSpace game engine. A paper summarizing some of this work, which involved an integration of PLN inference, OpenCogPrime:PerceptualPatternMining, and PredicateSchematization, is here XXX.

1. the Novamente Pet Brain, a specialized incarnation of the NCE, has been used (on an experimental, not yet productized basis) to control intelligent virtual pets in the Second Life and Multiverse virtual worlds. Some papers describing aspects of this work are at XXX and XXX.

Perception Processing in OpenCog Prime

Some detailed thoughts on the processing of perceptual data, and the generation of actions, in OpenCog Prime are at OpenCogPrime:PerceptionAction

Goal Oriented Cognition and World Interaction in OpenCog Prime

A detailed account of the "perception-cognition-action loop" via which OpenCog Prime is intended to interact with a world (physical or virtual) is at OpenCogPrime:GoalOrientedCognition.

Perceptual Pattern Mining

Finally, this section outlines an approach to perceptual pattern mining, designed for application in virtual worlds in which objects are represented in terms of polygons; but also with a view toward extensibility for more general usage.

The perceptual pattern mining problem is conceived as: Given the stream of perceptual data coming in from virtual-world, transform it into a series of predicates that are directly useful for inference in creating reasonably small inference trails that lead to results such as useful goal refinement, or useful schemata. This involves recognizing recurring patterns among perceptual data, in a manner guided by basic relationships of spatial and temporal proximity.

Modes of Visual Perception in Virtual worlds

The most complex, high-bandwidth perceptual data coming in from typical virtual world is visual data, and the treatment of visual data is a matter of some subtlety. There are basically three modes in which virtual-world may present visual data to OCP (or any other system):

- Object vision: OCP receives information about polygonal objects and their colors, textures and coordinates (each object is a set of contiguous polygons, and sometimes objects have "type" information, e.g. cube or sphere)

- Polygon vision: OCP receives information about polygons and their colors, textures and coordinates
- Pixel vision: OCP receives information about pixels and their colors and coordinates

In each case, coordinates may be given either in "world coordinates" or in "relative coordinates" (relative to the gaze). This distinction is not a very big deal since within an architecture like OCP, supplying schemata for coordinate transformation is trivial; and, even if treated as a machine learning task, this sort of coordinate transformation is not very difficult to learn. Our current approach is to prefer relative coordinates, as this approach is more natural in terms of modern Western human psychology; but we note that in some other cultures world coordinates are preferred and considered more psychologically natural.

At time of writing (July 2008) we have not done any work with pixel vision. We consider object vision to be too high-level for some purposes, and have settled on either polygon vision or a combination of polygon vision and object vision as the "right" approach for early AGI experiments in a virtual worlds context. The problem with object vision is that it removes the possibility for OCP to understand object segmentation. If, for instance, OCP perceives a person as a single object, then how can it recognize a head as a distinct sub-object? Feeding the system a pre-figured hierarchy of objects, sub-objects and so forth seems inappropriate in the context of an experiential learning system. On the other hand, the use of polygon vision instead of pixel vision seems to meet no such objections.

Beyond Generic Perceptual Pattern Mining

What is described in this section is a *generic* perceptual pattern mining approach for the virtual world context. However, it is not an exhaustive and complete approach to OCP perception, for two reasons

- In order to achieve adequate computational efficiency in perceptual processing in a complex virtual-world environment, it may be necessary to complement this general approach with a handful of additional, more highly specialized perceptual pattern mining subsystems. However, these will not be discussed significantly in this chapter, except in the following section which discusses object recognition.)
- The pattern mining approach described here is not intrinsically, algorithmically restricted to the virtual-world case, although it is presented in the virtual-world context. However, it is also not intended for usage in extremely high-bandwidth perception scenarios: e.g. for real-time streaming video data coming in from a camera eye. In situations like the latter, pattern mining as described here must still be done, but given currently realistic restrictions on processing speed, it should be on the output of more specialized lower-level pattern recognition algorithms. These also are not treated here, except briefly in these introductory remarks.

As an example of a specialized perceptual pattern mining subsystem, a preliminary design sketch has been made for an object recognition subsystem; thiis is presented in the following subsection. However, we don't consider this kind of "perception engineering" issue as extremely critical to the OCP AGI design: the key point is that the stochastic, greedy perceptual pattern mining approach is capable of integration with a variety of more specialized perceptual heuristics, which may be incorporated as necessary to maximize perceptual performance.

Regarding preprocessing to deal with high-bandwidth data, some thought has gone into how to preprocess visual data of the type obtainable by a contemporary robot to make it amenable to the kind of perceptual pattern mining process described here. One approach, for example, involves multiple phases:

- Use a camera eye and a lidar sensor in tandem, so as to avoid having to deal with stereo vision
- Using the above two inputs, create a continous 3D contour map of the perceived visual world
- Use standard mathematical transforms to polygon-ize the 3D contour map into a large set of small polygons

- Use heuristics to merge together the small polygons, obtaining a smaller set of larger polygons (but retaining the large set of small polygons for the system to reference in cases where a high level of detail is necessary)
- Feed the polygons into the perceptual pattern mining subsystem, analogously to the polygons that come in from virtual-world

In this approach, preprocessing is used to make the system see the physical world in a manner analogous to how it sees the virtual-world world.

Knowledge Representation for Perceptual Pattern Mining

Now we proceed toward describing how perceptual pattern mining may be carried out. In this subsection we discuss the representation of knowledge, and then in the following subsection we discuss the actual mining.

Input Data

First, we may assume that each sensation from virtual-world is recorded as set of "transactions", each of which is of the form

```
Time, 3D coordinates, object type
```

or

```
Time, 3D coordinates, action type
```

Each transaction may also come with an additional list of (attribute, value) pairs, where the list of attributes is dependent upon the object or action type. Transactions are represented at Atoms, and don't need to be a specific Atom type — but are referred to here by the special name *transactions* simply to make the discussion clear.

Next, define a transaction template as a transaction with location and time information set to *wild cards* — and potentially, some other attributes set to wild cards. (These are implemented in terms of Atoms involving VariableNodes.)

For instance, some transaction templates in the current virtual-world might be informally represented as:

```
Reward

Red cube

kick

move_forward

Cube

Cube, size 5

me

Teacher
```

Basic Spatiotemporal Predicates

Next, I define here a set of "basic spatiotemporal predicates". These predicates apply to tuples of transactions, and also (by averaging) to tuples of transaction templates.

For two transactions A and B with the same time-stamp (or more generally, very close-by time-stamps), predicates such as

```
near(A,B)
```

```
in_front_of(A,B)
```

should be calculated. These are assumed fuzzy predicates with truth values in [0,1] for each argument list.

Also, for two transactions A and B that are very near to each other in time (judged by some threshold), the predicates

```
shortly_after(A,B)
```

```
simultaneous(A,B)
```

should be calculated (again, fuzzy predicates). For convenience, in this chapter I will denote these relationships by

```
SSeqAND A B
```

```
SimAND A B
```

(where SSeqAND means *short sequential AND*, i.e. SeqAND with a short time lag.)

For instance if RedCubeObservation99 and TeacherObservation44 are specific transactions, then

```
near(RedCubeObservation99, TeacherObservation44)
```

```
SimAND (RedCubeObservation99, TeacherObservation44)
```

together assess the degree to which this observed red cube and the observed Teacher are near each other in space and time. On the other hand,

```
near(red cube, Teacher)
```

indicates the degree to which the red cube and the teacher are often near each other.

Later we may want other fuzzy predicates too, such as size, e.g.

```
cube_size(A,5)
```

would be truer for cubes A whose size is closer to 5, and maximally true of cubes of size 5.

And perhaps later, if A and B are very close in spatial location to each other and not extremely far from each other in time, we also want to compute

```
after(A,B)
```

(a different fuzzy predicate, with a different weighting on time lags).

Transaction Graphs

Next we may conceive a transaction graph, whose nodes are transactions and whose links are labeled with labels like after, SimAND, SSeqAND, near, in_front_of, and so forth (and whose links are weighted as well).

We may also conceive a transaction template graph, whose nodes are transaction templates, and whose links are the same as in the transaction graph.

And finally, we may conceive a transaction template relationship graph (TTRG), whose nodes may be any of: transactions; transaction templates; basic spatiotemporal predicates evaluated at tuples of transactions or transaction templates.

Spatiotemporal Conjunctions

Define a temporal conjunction as a conjunction involving SimultaneousAND and SequentialAND operators (including SSeqAND as a special case of SeqAND: the special case that interests us in the short term). The conjunction is therefore ordered, e.g.

A SSeqAND B SimAND C SSeqAND D

We may assume that the order of operations favors SimAND, so that no parenthesizing is necessary.

Next, define a basic spatiotemporal conjunction as a temporal conjunction that conjoins terms that are either

- transactions, or
- transaction templates, or
- basic spatiotemporal predicates applied to tuples of transactions or transaction templates

I.e. a basic spatiotemporal conjunction is a temporal conjunction of nodes from the transaction template relationship graph.

An example would be:

```
(hold ball) SimAND ( near(me, teacher) ) SSeqAND Reward
```

[this assumes that the *hold* action has an attribute that is the type of object held, so that

```
hold ball
```

in the above temporal conjunction is a shorthand for the transaction template specified by

```
action type: hold

object_held_type: ball
```

This example says that if the agent is holding the ball and is near the teacher then shortly after that, the agent will get a reward.

The Mining Task

The perceptual mining task, then, is to find basic spatiotemporal conjunctions that are *interesting.* What constitutes interestingness is multifactorial, and includes.

- involves important Atoms (e.g. Reward)
- has a high temporal cohesion (i.e. the strength of the time relationships embodied in the SimAND and SeqAND links is high)
- has a high spatial cohesion (i.e. the near() relationships have high strength)
- has a high frequency
- has a high surprise value (its frequency is far from what would be predicted by its component sub-conjunctions

Note that a conjunction can be interesting without satisfying all these criteria; e.g. if it involves something important and has a high temporal cohesion, we want to find it regardless of its spatial cohesion.

For starters, for simple virtual-world experiments, defining interestingness as the combination frequency and temporal cohesion should be adequate.

A Mining Approach

One tractable approach to perceptual pattern mining is greedy and iterative, involving the following steps:

1. Build an initial transaction template graph G
2. Greedily mine some interesting basic spatiotemporal conjunctions from it, adding each interesting conjunction found as a new node in G (so that G becomes a transaction template relationship graph), repeating step 2 until boredom results or time runs out

The same TTRG may be maintained over time, but of course will require a robust forgetting mechanism once the history gets long or the environment gets nontrivially complex.

The greedy mining step may involve simply grabbing SeqAND or SimAND links with probability determined by the (importance and/or interestingness) of their targets, and the probabilistic strength and temporal strength of the temporal AND relationship, and then creating conjunctions based on these links (which then become new nodes in the TTRG, so they can be built up into larger conjunctions).

The Perceptual-Motor Hierarchy

The perceptual approach outlined in OpenCogPrime:PerceptualPatternMining is "flat," in the sense that it simply proposes to recognize patterns in a stream of perceptions, without imposing any kind of explicitly hierarchical structure on the pattern recognition process or the memory of perceptual patterns. This is different

from how the human visual system works, with its clear hierarchical structure, and also different from many contemporary vision architectures, such as Hawkins' Numenta system which is a hierarchical neural networks.

However, the approach described above may be easily hierarchalized within the OCP architecture, and this is likely the most effective way to deal with complex visual scenes. Most simply, in this approach, a hierarchy may be constructed corresponding to different spatial regions, within the visual field. The RegionNodes at the lowest level of the hierarchy correspond to small spatial regions, the ones at the next level up correspond to slightly larger spatial regions, and so forth. Each RegionNode also correspond to a certain interval of time, and there may be different RegionNodes corresponding to the same spatial region but with different time-durations attached to them. RegionNodes may correspond to overlapping rather than disjoint regions.

Within each region mapped by a RegionNode, then, perceptual pattern mining as defined in the previous section may occur. The patterns recognized in a region are linked to the corresponding RegionNode — and are then fed as inputs to the RegionNodes corresponding to larger, encompassing regions; and as suggestions-to-guide-pattern-recognition to nearby RegionNodes on the same level. This architecture involves the fundamental hierarchical structure/dynamic observed in the human visual cortex. Thus, the hierarchy incurs a dynamic of patterns-within-patterns-within-patterns, and the heterarchy incurs a dynamic of patterns-spawning-similar-patterns.

Also, patterns found in a RegionNode should be used to bias the pattern-search in the RegionNodes corresponding to smaller, contained regions: for instance, if many of the sub-regions corresponding to a certain region have revealed parts of a face, then the pattern-mining processes in the remaining sub-regions may be instructed to look for other face-parts.

This architecture permits the hierarchical dynamics utilized in standard hierarchical vision models, such as Jeff Hawkins' and other neural net models, but within the context of OCP's pattern-mining approach to perception. It is a good example of the flexibility intrinsic to the OCP architecture.

Finally, why have we called it a perceptual-*motor* hierarchy above? This is because, due to the embedding of the perceptual hierarchy in OCP's general Atom-network, the percepts in a certain region will automatically be linked to actions occurring in that region. So, there may be some perception-cognition-action interplay specific to a region, occurring in parallel with the dynamics in the hierarchy of multiple regions. Clearly this mirrors some of the complex dynamics occurring in the human brain.

Identifying Objects from Polygonal Perceptions

This section describes a strategy for identifying objects in a visual scene that is perceived as a set of polygons. It is not a thoroughly detailed algorithmic approach, but rather a high-level description of how this may be done effectively within the OCP design. It is offered here largely as an illustration of how specialized perceptual data processing algorithms may be designed and implemented within the OCP framework.

We deal here with an agent whose perception of the world, at any point in time, is understood to consist of a set of polygons, each one described in terms of a list of corners. The corners may be assumed to be described in coordinates relative to the viewing eye of the agent.

What I mean by "identifying objects" here is something very simple. I don't mean identifying that a particular object is a chair, or is Ben's brown chair, or anything like that — I simply mean identifying that a given collection of polygons is meaningfully grouped into an object. That is the task considered here.

Of course, not all approaches to polygon-based vision processing would require this sort of phase: it would be possible, as an alternative, to simply compare the set of polygons in the visual field to a database of prior experience and then do object identification (in the present sense) based on this database-comparison. But in the approach described in this page, we have chosen not to take this approach, and to begin instead with an automated segmentation of the set of perceived polygons into a set of objects.

Algorithm Overview

The algorithm described here falls into three stages:

1. Recognizing PersistentPolygonNodes (PPNodes) from PolygonNodes.
2. Creating Adjacency Graphs from PPNodes.
3. Clustering in the Adjacency Graph.

Each of these stages involves a bunch of details, not all of which have been fully resolved: this document just gives a conceptual overview.

I will speak in terms of objects such as PolygonNode, PPNode and so forth, because inside the OCP AI engine, observed and conceived entities are represented as nodes in an graph. However, this terminology is not very important here, and what I call a PolygonNode here could just as well be represented in a host of other ways, within the overall OCP framework.

Recognizing PersistentPolygonNodes (PPNodes) from PolygonNodes

A PolygonNode represents a polygon observed at a point in time. A PPNode represents a series of PolygonNodes that are heuristically guessed to represent the same PolygonNode at different moments in time.

Before "object permanence" is learned, the heuristics for recognizing PPNodes will only work in the case of a persistent polygon that, over an interval of time, is experiencing relative motion within the visual field, but is never leaving the visual field. For example some reasonable heuristics are: If P1 occurs at time t, P2 occurs at time s where s is very close to t, and P1 are similar in shape, size and color and position, then P1 and P2 should be grouped together into the same PPNode.

More advanced heuristics would deal carefully with the case where some of these similarities did not hold, which would allow us to deal e.g. with the case where an object was rapidly changing color.

In the case where the polygons are coming from a simulation world like AGISIM, then from our positions as programmers and world-masters, we can see that what a PPNode is supposed to correspond to is a certain side of a certain AGISIM object; but it doesn't appear immediately that way to OCP when controlling an agent in AGISIM since NM isn't perceiving AGISIM objects, it's perceiving polygons. On the other hand, in the case where polygons are coming from software that postprocesses the output of a Lidar based vision system, then the piecing together of PPNodes from PolygonNodes is really necessary.

Creating Adjacency Graphs from PPNodes

Having identified PPNodes, we may then draw a graph between PPNodes (called an "Adjacency Graph"), wherein the links are AdjacencyLinks (with weights indicating the degree to which the two PPNodes tend to be adjacent, over time). [Note for later: A more refined graph might also involve SpatialCoordinationLinks (with weights indicating the degree to which the vector between the centroids of the two PPNodes tends to be consistent over time).]

We may then use this graph to do object identification:

- First-level objects may be defined as clusters in the graph of PPNodes.
- One may also make a graph between first-level objects, an ObjectGraph with the same kinds of links as in the PPGraph. Second-level objects may be defined as clusters in the ObjectGraph.

The "strength" of an identified object may be assigned as the "quality" of the cluster (measured in terms of how tight the cluster is, and how well separated from other clusters.)

As an example, consider a robot with two parts: a body and a head. The whole body may have a moderate strength as a first-level object, but the head and body individually will have significantly greater strengths as first-level objects. On the other hand, the whole body should have a pretty strong strength as a second-level object.

It seems convenient (though not necessary) a PhysicalObjectNode type to represent the objects recognized via clustering; but the first versus second level object distinction should definitely not be made on the Atom type level.

Building the adjacency graph requires a mathematical formula defining what it means for two PPNodes to be adjacent. Creating this formula may require a little tuning. For instance, the adjacency between two PPNodes PP1 and PP2 may be defined as the average over time of the adjacency of the PolygonNodes PP1(t) and PP2(t) observed at each time t. (A p'th power average may be used here, and different values of p may be tried.) Then, the adjacency between two (simultaneous) PolygonNodes P1 and P2 may be defined as the average over all x in P1 of the minimum over all y in P2 of sim(x,y), where sim(,) is an appropriately scaled similarity function. This latter average could arguably be made a maximum; my inclination would be to make it a p'th power average with large p, which approximates a maximum.

Clustering in the Adjacency Graph.

As noted above, the idea is that objects correspond to clusters in the adjacency graph. This means we need to implement some hierarchical clustering algorithm that is tailored to find clusters in symmetric weighted graphs. Probably some decent algorithms of this character exist, if not it would be fairly easy to define one, e.g. by mapping some standard hierarchical clustering algorithm to deal with graphs rather than vectors.

Clusters will then be mapped into AGISIMObjectNodes, interlinked appropriately via PhysicalPartLinks and AdjacencyLinks. (E.g. there would be a PhysicalPartLink between the AGISIMObjectNode representing a head and the AGISIMObjectNode representing a body (where the body is considered as including the head)).

Discussion

It seems probable that, for simple scenes consisting of a small number of simple objects, clustering for object recognition will be very unproblematic. However, there are two cases that seem potentially tricky:

- Sub-objects: e.g. the head and torso of a body, which may move separately; or the nose of the head, which may wiggle; or the legs of a walking dog; etc.
- Coordinated objects: e.g. if a character's hat is on a table, and then later on his head, then when it's on his head we basically want to consider him and his hat as the same object, for some purposes.

These examples show that partitioning a scene into *objects* is a borderline-cognitive rather than purely lower-level-perceptual task, which cannot be hard-wired in any very simple way.

I also note that, for complex scenes, clustering may not work perfectly for object recognition and some reasoning may be needed to aid with the process. Intuitively, these may correspond to scenes that, in human perceptual psychology, require conscious attention and focus in order to be accurately and usefully perceived.

Perception, Action, and Multiple Interaction Channels

Let us now take a look at OCP perception and action from the perspective of software architecture and cognitive architecture. To understand OCP as an embodied perceiving and acting system, let's start with the external world. External-world events come into the OCP system from physical or virtual world sensors, plus from other sources such as database interfaces, Web spiders, and/or other sources. The external systems providing OCP with data may be generically referred to as sensory sources. Once Atoms have been created to represent external data, then one is dealing with perceptions rather than sensations.

What we call an "interaction channel" is a collection of sensory sources that is intended to be considered as a whole as a synchronous stream, and that is also able to receive OCP actions — in the sense that when OCP carries out actions relative to the interaction channel, this directly affects the perceptions that OCP receives from the interaction channel.

A OCP meant to have conversations with 10 separate users at once might have 10 interaction channels. A human mind has only one interaction channel in this sense (although humans may become adept at processing information from multiple external-world sources, coming in through the same interaction channel).

Multiple-interaction-channel digital psychology may become extremely complex — and hard for us, with our single interaction channels, to comprehend. This is one among many cases where a digital mind, with its more flexible architecture, will have a clear advantage over our human minds with their fixed and limited neural architectures. For simplicity, however, in the following chapters we will often focus on the single-interaction-channel case.

Events coming in through an interaction channel are presented to the system as new perceptual Atoms, such as PixelNodes, PolygonNodes, CharacterInstanceNodes or NumberInstanceNodes, and relationships amongst these. The AttentionValues of these newly created Atoms require special treatment. Not only do they require special rules, they require additional fields to be added to the AttentionValue object, beyond what has been discussed so far.

We require newly created perceptual Atoms to be given a high initial STI. And we also require them to be given a high amount of a quantity called "interaction-channel STI." To support this, the AttentionValue objects of Atoms must be expanded to contain interaction-channel STI values; and the ImportanceUpdating CIMDynamic must apply the IUF to compute interaction-channel importance separately from ordinary importance.

Some interaction channels will come with mechanisms that give certain new perceptual Atoms more generic and interaction-channel-specific STI than others.

Newly created perceptual Atoms must also have their TruthValues set, with strength=1 indicating that the Atom denotes something definitely present in the observed world, and strength < 1 denoting uncertain presence. The first-order inference CIM-Dynamics will then propagate these strength variables, via the node probability inference rules. Via this inference process, for example, when a ListLink corresponding to the word *cow* is created, the corresponding ConceptNode *cow* will gain additional strength.

Just as we have channel-specific AttentionValues, we may also have channel-specific TruthValues. This allows the system to separately account for the frequency of a given perceptual item in a given interaction channel. However, no specific mechanism is needed for these, they are merely contextual truth values, to be interpreted within a Context Node associated with the interaction channel.

Once perceptual Atoms have been created, various perceptual CIMDynamics comes into play, taking perceptual schemata (schemata whose arguments are perceptual nodes or relations therebetween) and applying them to Atoms recently created (creating appropriate ExecutionLinks to store the results). The need to have special, often modality-specific perception CIMDynamics to do this, instead of just leaving it to the generic SchemaExecution CIMDynamic, has to do with computational efficiency, scheduling and parameter settings. Perception CIMDynamics are doing schema execution urgently, and it's doing it with parameter settings tuned for perceptual processing. This means that, except in unusual circumstances, newly received stimuli will be processed immediately by the appropriate perceptual schemata.

Some newly formed perceptual Atoms will have links to existing atoms, ready-made at their moment of creation. CharacterInstanceNodes and NumberInstanceNodes are examples; they are born linked to the appropriate CharacterNodes and NumberNodes. Of course, atoms representing perceived relationships, perceived groupings, etc., will not have ready-made links and will have to grow such links via the standard cognitive processes. The context formation CIMDynamic looks at perceptual atom creation events and creates Context Nodes accordingly; this should be timed so that the Context Nodes are entered into the system rapidly, so that they can be used by the processes doing initial-stage link creation for new perceptual atoms.

As discussed above, in a full OCP configuration, newly created perceptual nodes and perceptual schemata will reside in a special perception-oriented Units. The purpose of this is to ensure that perceptual processes occur rapidly, not delayed by slower cognitive processes.

Finally, separate from the ordinary perception process, it is also valuable for there to be a direct route from the system's sensory sources to the overall MindDB that records all of the system's experience. This does not involve perceptual schemata at all, nor is it left up to the sensory source; rather, it is carried out by the OCP server at the point where it receives input from the sensory source. This experience database is a record of what the system has seen in the past, and may be mined by the system in the future for various purposes. The creation

of new perceptual atoms should also be stored in the experience database; this portion of the MindDB is what the "TimeServer" process mentioned above accesses, in the context of context formation.

Obviously, the MindDB is something that has no correlate in the human mind/brain. This is a case where OCP takes advantage of the non-brainlike properties of its digital computer substrate. The OCP perception process is intended to work perfectly well without access to the comprehensive database of experiences potentially stored in the MindDB. However, a complete record of a mind's experience is a valuable thing, and there seems no reason for the system not to exploit it fully. Advantages like this allow the OCP system to partially compensate for its lack of some of the strengths of the human brain as an AI platform, such as massive parallelism.

Internal Simulations

This brief page deals with an important cognitive component of OCP: the component concerned with creating *internal simulations* of situations in the external physical world.

This is a component that is likely significantly different in OCP from anything that exists in the human brain, yet, the function that it carries out is obviously essential to human cognition; indeed, more so to human cognition than to OCP's cognition, because OCP is by design more reliant on formal reasoning than the human brain is.

Much of human thought consists of internal, quasi-sensory "imaging" of the external physical world. Often this takes the form of visualization, but not always. Blind people think in terms of internal imagery, and many sighted people think in terms of sounds, tastes or smells in addition to visual images.

So far, the various mechanisms proposed as part of OCP do not have much to do with this kind of internal imagery, that seems to play such a large role in human thought. This is OK, of course, since OCP is not intended as a simulacrum of human thought, but rather as a different sort of intelligence.

However, we believe it will actually be valuable to OCP to incorporate a sort of inner imagery. And for that purpose, we propose a novel mechanism: the incorporation within the OCP system of a 3D physical-world simulation engine.

The current use of virtual worlds for OpenCog is to provide a space in which human-controlled agents and OCP-controlled agents can interact, thus allowing flexible instruction of the OCP system by humans, and flexible embodied, grounded learning by OCP systems. But this very same mechanism may be used internally to OCP, i.e. a OCP system may be given an *internal simulation world*, which serves as a sort of "mind's eye." Any sufficiently flexible virtual world software may be used for this purpose, for example OpenSim or RealXTend.

Atoms encoding percepts may be drawn from memory and used to generate forms within the internal simulation world. These forms may then interact according via

- forms acting according to the patterns via which they are remembered to act
- the laws of physics, as embodied in the simulation world

This allows a kind of "implicit memory," in that patterns emergent from the world-embedded interaction of a number of entities *need not explicitly be stored in memory*, so long as they will emerge when the entities are re-awakened within the internal simulation world.

The SimulatorMindAgent grabs important perceptual Atoms and uses them to generate forms within the internal simulation world, which then act according to remembered dynamical patterns, with the laws of physics filling in the gaps in memory. This provides a sort of running internal visualization of the world. Just as important, however, are specific schemata that utilize visualization in appropriate contexts. For instance, if reasoning is having trouble solving a problem related to physical entities, it may feed these entities to the internal simulation world to see what can be discovered. Patterns discovered via simulation can then be fed into reasoning for further analysis.

The process of perceiving events and objects in the simulation world is essentially identical to the process of perceiving events and objects in the "actual" world.

And of course, an internal simulation world may be used whether the OCP system in question is hooked up to a simulation world like AGISim, or to a physical robot.

Finally, perhaps the most interesting aspect of internal simulation is the generation of "virtual perceptions" from abstract concepts. Analogical reasoning may be used to generate virtual perceptions that were never actually perceived, and these may then be visualized. The need for "reality discrimination" comes up here, and is easier to enforce in OCP than in humans. A PerceptNode that was never actually perceived may be explicitly embedded in a HypotheticalLink, thus avoiding the possibility of confusing virtual percepts with actual ones. How useful the visualization of virtual perceptions will be to OCP cognition, remains to be seen. This kind of visualization is key to human imagination but this doesn't mean it will play the same role in OCP's quite different cognitive processes. But it is important that OCP has the power to carry out this kind of imagination.

Chapter Ten: Goal-Oriented Cognition

Goal-Oriented Cognition

Not all of OCP's cognition is driven by goals, but much of it is. The pages supervened by this one:

- **OpenCogPrime:GoalDrivenCognition**
- OpenCogPrime:UberGoalDynamics
- OpenCogPrime:SchemaContextGoalTriads
- OpenCogPrime:GoalAtoms
- OpenCogPrime:ContextAtoms
- OpenCogPrime:FeelingNodes
- OpenCogPrime:GoalFormation
- OpenCogPrime:GoalFulfillmentAndPredicateSchematization
- OpenCogPrime:ContextFormation
- OpenCogPrime:ExecutionManagement
- OpenCogPrime:GoalsAndTime
- OpenCogPrime:EconomicGoalAndActionSelection

review the details of goal-oriented cognition in OCP — a conceptually fairly straightforward, but technically quite complex topic, which interweaves all the other aspects of the OCP design.

We describe, in this page and the ones it supervenes over, OCP as an embodied, goal-driven, autonomous learning system. This material provides the conceptual and dynamical framework within which the cognitive mechanisms described in other pages of the OCP documentation operate.

The page OpenCogPrime:PredictiveAttraction reminds you of some necessary structures in temporal logic, important for interpreting some of the above-linked pages.

These cognitive mechanisms must be understood, in a OCP context, not as algorithms for solving isolated problems, but as components of a complex system achieving complex goals in complex environments. The interactions between the mechanisms will be discussed more in later chapters; here we will focus on their interoperation in the context of explicitly goal-seeking behavior.

XX insert from PPT basic diagram of world, perception, action, goals, feelings... XX

The above diagram loosely summarizes the sort of *high-level control system* that we believe is critical to intelligence. The parts of the diagram are generically meaningful and we propose that any intelligent system operating under limited resources must explicitly manifest these processes in some way. We will also indicate the types of OCP Atoms centrally or uniquely involved with each function.

Many but not all of the ideas in this chapter were implemented in the NCE in simplified form, in 2005, in the context of using the NCE to control a humanoid agent in the AGISim simulation world.

Goal, Feelings and Contexts

The pages supervened by this one largely involve three key concepts — Contexts, Goals and Feelings — and their interrelationships, and their (interdependent) roles in OCP system dynamics.

As it turns out, none of these concepts requires fundamentally new data structures. Rather, they may all be understood as *wrappers* around structures already existing in OCP for other purposes.

Goals do however involve a largely new process: the process of OpenCogPrime:EconomicGoalSelection, an outgrowth of economic attention allocation that is treated on another page.

Philosophically, we may say that, in the OCP conceptual framework, goals, feelings and contexts are represented as particular manifestations of deeper, more generic cognitive structures and processes.

Cognitive Processes Related to Contexts & Goals

Some unique cognitive processes associated with Goal and Context Atoms, reviewed in various of the pages supervened by this one, are:

- OpenCogPrime:ContextFormation — the creation of ContextLinks treating as Contexts states frequently experienced by OCP. This is done by a special instance of the OpenCogPrime:MapEncapsulation

MindAgent, with parameter values tuned for context formation; and by the ConceptCreation and Clustering MindAgents.
- OpenCogPrime:GoalFormation — the creation of Goal Atoms representing goals that OCP wants to attain. The goal formation CIMDynamic, relying on PLN, seeks to form Goal Nodes that imply important and long-term-important Goal Atoms. Of course, this process is initially driven by the initial set of given Goal Atoms.
- OpenCogPrime:SchemaContextGoalTriad formation — carried out by PLN inference via a specialized control mechanism called *predicate schematization*, which is specially tuned to look for this kind of implication
- OpenCogPrime:GoalFulfillmentAndPredicateSchematization — the process by which a goal seeks schemata that it judges will cause its satisfaction, and activates them.
- OpenCogPrime:EconomicGoalAndActionSelection — the process by which some goals are chosen to be allowed to actively seek fulfillment, at a certain point in time; and actions are adaptively selected to fulfill these goals

Contexts/Feelings/Goals and Time

As with other Atoms, when a Context, Feeling or Goal Atom's TruthValue is evaluated at a certain time point, the value obtained is then merged in with the old value according to the revision rule. The temporal reweighting process is very important here, so that TruthValue reflects recent goal/context satisfaction more so than ancient satisfaction. A key point is that some Context Nodes and Goal Nodes must have a very high time decay factor, as compared to ordinary Predicates, because one sometimes wants the truth value of a Context Node to represent whether a context is relevant right now, not whether it was relevant last week.

Context, Goals and Importance

Oftentimes, part of a Goal or a Context will be a Link specifying that *Atoms so and such are important in the system* or *Atoms so and such are important in interaction channel C.* (A little later in this chapter, we'll review the notion of interaction-channel-dependent importance.) To achieve this the Goal or Context Atoms will contain Links of the form

```
HypotheticalLink (ExecutionLink Importance A x)

HypotheticalLink (ExecutionLink (InteractionChannelImportance C) A x)
```

where A is an Atom and x is a number.

Ubergoal Dynamics

In the early phases of a OCP system's cognitive development, the goal system dynamics will be quite simple. The ubergoals are supplied by human programmers, and the system's adaptive cognition is used to derive subgoals. Attentional currency allocated to the ubergoals is then passed along to the subgoals, as judged appropriate.

As the system becomes more advanced, however, more interesting phenomena may arise regarding ubergoals: implicit and explicit ubergoal creation.

Implicit Ubergoal Pool Modification

First of all, *implicit ubergoal creation or destruction* may occur. Implicit ubergoal destruction may occur when there are multiple ubergoals in the system, and some prove easier to achieve than others. The system may then decide not to bother achieving the more difficult ubergoals. Appropriate parameter settings may mitigate against this phenomenon, of course.

Implicit ubergoal creation may occur if some Goal Node G arises that inherits as a subgoal from multiple ubergoals. This Goal G may then come to act implicitly as an ubergoal, in that it may get more attentional currency than any of the ubergoals.

Also, implicit ubergoal creation may occur via forgetting. Suppose that G becomes a goal via inferred inheritance from one or more ubergoals. Then, suppose G forgets *why* this inheritance exists, and that in fact the reason becomes obsolete, but the system doesn't realize that and keeps the inheritance there. Then, G is an implicit ubergoal in a strong sense: it gobbles up a lot of attentional currency, potentially more than any of the actual ubergoals, but actually doesn't help achieve the ubergoals, even though the system thinks it does. This kind of dynamic is obviously very bad and should be avoided — and *can* be avoided with appropriate tuning of system parameters (so that the system pays a lot of attention to making sure that its subgoaling-related inferences are correct and are updated in a timely way).

Explicit Ubergoal Pool Modification

An advanced OCP system may be given the ability to explicitly modify its ubergoal pool. This is a very interesting but very subtle type of dynamic, which is not currently well understood — and which potentially could be very dangerous, in the case of an intelligent and capable OCP instance.

However, modification, creation and deletion is a key aspect of human psychology, and the granting of this capability to mature OCP systems must be seriously considered.

In the case that ubergoal pool modification is allowed, one useful heuristic may be to make *implicit ubergoals* into explicit ubergoals. For instance: if an Atom is found to consistently receive a lot of RFSs, and has a long time-scale associated with it, then the system should consider making it an ubergoal. But this heuristic is certainly not the whole story, and any advanced OCP system that is going to modify its own ubergoals should definitely be tuned to put a lot of thought into the process!

The science of ubergoal pool dynamics basically does not exist at the moment, and one would like to have some nice mathematical models of the process prior to experimenting with it in any intelligent, capable OCP system. (Students of advanced AGI Ethics may fairly consider the prior sentence an exercise in exorbitant understatement.)

Context-Schema-Goal Triads

A particularly critical aspect of intelligence, we propose, is the *Context-Schema-Goal triad*: a context, schema and goal that are interlinked according to the general pattern

enactment of the schema S in the context C is likely to result in the attainment of the goal state G.

Technically, this means:

```
Context

    C

    PredictiveAttraction

        Evaluation done S

        G
```

(here done is a built-in schema which basically indicates that S has been executed on some input), or

```
Context

    C

    PredictiveAttraction

        Execution S A *

        G
```

where A is some particular input.

In OCP, sometimes a context-schema-goal triad will be represented on the Atom level, sometimes on the map level. These triads form the essence of OCP's *behavioral reinforcement learning* subsystem, which allows the system to learn new behaviors through experience. These *behaviors* include cognitive behaviors — i.e. the learning of the *cognitive schemata* that have been discussed frequently in previous chapters.

Goal Atoms

A Goal Atom represents a *target system state* and is true to the extent that the system satisfied the conditions it represents. A Context Atom represents an *observed state of the world/mind,* and is true to the extent that the state it defines is observed. Taken together, these two Atom types provide the infrastructure OCP needs to orient its actions in specific contexts toward specific goals. Not all of OCPís activity is guided by these Atoms ñ much of it is non-goal-directed and spontaneous, or *ambient* as we sometimes call it. But it is important that some of the systemís activity ñ and in some cases, a substantial portion ñ is controlled explicitly via goals.

Specifically, a Goal Atom is simply an Atom (usually a PredicateNode, sometimes a Link, and potentially another type of Atom) that has been selected by the GoalRefinement CIM-Dynamic as one that represents a state of the atom space which the system finds important to achieve. The extent to which an Atom is considered a Goal Atom at a particular point in time is determined by how much of a certain kind of financial instrument called an RFS it possesses. The logic of RFS's will be explained in a later section in this chapter.

A OCP instance must begin with some initial *ubergoals* (aka *top level supergoals*), but may then refine these goals in various ways using inference. Immature, "childlike" OCP systems cannot modify their ubergoals nor

add nor delete ubergoals. Advanced OCP systems may be allowed to modify, add or delete ubergoals, but this is a critical and subtle aspect of system dynamics that must be treated with great care.

Context Atoms

Next, a Context is simply an Atom that is used as the source of a PLN ContextLink, for instance

```
Context

    quantum_computing

    Inheritance Ben amateur
```

or

```
Context

    game_of_fetch

    PredictiveAttraction

        Evaluation give (ball, teacher)

        Satisfaction
```

The former simply says that Ben is an amateur in the context of quantum computing. The latter is a more pertinent example to the present page: it says that in the context of the game of fetch, giving the ball to the teacher implies satisfaction.

Feeling Nodes

The general concept of a FeelingNode is that of an *internal sensor* — a FeelingNode is a mechanism that a OCP system uses to sense some aspect of itself. FeelingNodes are not intended to represent emotions, which we understand as complex distributed phenomena, not encapsulable in single nodes. There are some parallels and relationships between FeelingNodes and emotions, which will arise sometimes in our discussion of FeelingNodes, but we don't intend these parallels to be taken very seriously. The study of emotions in AGI systems will be a subtle science, we believe, but one that will be difficult to seriously explore prior to the development of powerful AGI systems that can communicate fluently with humans.

Like Goal and Context Atoms, FeelingNodes, structurally, are simply PredicateNodes. Specifically, they are a subclass of OpenCogprime:GroundedPredicateNode), which is grounded in a Procedure that measures either

- some sort of global indicator of the state of the system, or
- an in-built feeling indicator, such as the Pleasure FeelingNode, which is activated via things such as getting rewarded by the teacher

More technically, they are PredicateNodes whose internal schema expression trees have at their leaves (along with, perhaps, other inputs) either

- FeelingSchemata, or
- OutputValues of FeelingNodes

A set of elementary FeelingSchemata is defined; some FeelingNodes may simply wrap up a single FeelingSchema. Others, containing combinations of elementary FeelingSchema are called compound FeelingNodes.

A FeelingNode's truth value will vary over time, in accordance with the variance of the output of the elementary FeelingSchemata it's defined in terms of.

Learning new FeelingSchemata and modifying or deleting existing ones are advanced system dynamics, which may be undertaken only by advanced OCP systems that are able to learn procedures based on complex experience-based inferences.

Note that the importance of a FeelingNode bears no direct relationship to its truth value.

For instance, the *Satisfaction FeelingNode* (representing the degree to which the system's goals are satisfied, overall) may have a high importance but a low truth value — this might (very loosely speaking) be considered a variety of *depression.* Also, FeelingNodes may be embedded in Goal Atoms, with either positive or negative valence. One could have a goal of being satisfied (having a high truth value for the Satisfaction FeelingNode), a goal of being not overworked (having a low truth value for the SystemStress FeelingNode), etc.

The key thing distinguishing elementary feelings from compound feelings is that elementary feelings are evaluable completely automatically, without any thought or any adaptation based on experience. From the experiencing OCP mind's point of view, these feelings simply are what they are. Only once OCP reaches the point of analyzing and modifying its source code, will it have control over its elementary feelings.

However, any OCP may have *control* over the compound feelings it generates. This leads to some interesting philosophical and psychological issues. The notion of *control* becomes bound up with the Consciousness distinction and the problem of will. When we say a OCP has control over the compound FeelingNodes it generates and utilizes, we mean that its mental processes may adapt these based on its experience, either in service of its goals or spontaneously. Whether it feels subjectively that it has control over these FeelingNodes and associated dynamics, is a different question entirely — which gets into the complex issue of the nature of emotions in an non-evolved, nonhuman intelligence. One thing that can potentially happen is that FeelingNodes are extensively created and modified and utilized via non-goal-oriented self-organizing dynamics, and goal-satisfaction-oriented dynamics fail to exert significant influence over them. In this case, the system could rationally come to model itself as being unable to achieve its goals because its feelings were *running amok.*

A Provisional List of Elementary Feelings

Here we will define a set of initial, elementary feelings for OCP. A set of initial, elementary goals will follow from these. The initial feeling set described here is the one we used for initial experimentation in prototypes, but it's very much a *first draft* — we envision that further work will lead to substantial modifications and additions.

TeacherSatisfaction — How happy are the OCP instance's human friends with what the system is doing?

This may be gauged in several possible ways depending on the OCP front end in use:

- explicit user responses to system outputs (e.g., the user clicking a reward button, in the AGISim user interface)
- interpretation of the user's natural language responses (*Good boy, OCP!*)
- objective measures of user response: how long does the user view the results returned by a OCP-powered search engine, etc.

Gaining Understanding — How much new pattern is being formed in the system overall?

One crude way to assess this is: what is the total s*d value of set new atoms, normalized by the total size of the atoms in this set? (The normalization by size is only needed because of the existence of atoms containing compound schema and predicate expressions.) This is an important system goal: the system is supposed to want to learn things!

External Novelty — How much is the perceived world of the system changing over time?

A love for novelty encourages exploration which broadens the mind.

Internal Novelty — How much is the internal landscape changing over time, the Atomspace itself?

Health — a combination of a number of elementary indicators to give a composite indicator of health.

These indicators may include such things as free memory and response time for certain types of queries submitted by users. One may also create individual FeelingSchema for these elementary indicators, such as FreeSystemMemory or QueryResponseTime (in a OCP-driven query-oriented software application), etc. — a strategy which allows OCP to construct its own composite Health feelings.

Satisfaction — a compound feeling which is a combination (for example, a weighted average) of the feelings mentioned above.

This is the most critical FeelingNode, as it is the basis of OCP's motivational system. Of course, the use of the English word *Satisfaction* is not entirely accurate here; but we've found that any term with any evocative power also has the power to mislead, and after rejecting *Happiness* as being too simplistically misleading, we've settled on *Satisfaction.*

What makes *Satisfaction* OCP's primary motivator is simply the fact that the Goal Node referring to the goal of having high Satisfaction (i.e. of the FeelingNode having a high truth value) is constantly *autonomically* given a high Short-Term Importance. This causes a large percentage of the processes of the system to focus on the Satisfaction FeelingNode in various direct and indirect ways.

This initial feeling framework is intentionally very simplistic. The intention is that more complex feelings should emerge via the natural evolutionary processes of the system. Imposing a subtle system of human-like feelings on OCP is probably not desirable. Rather, we should give it a very simple framework for sensing itself, and allow higher-level emotions to emerge as is appropriate for the system given its particular goals and contexts.

Goal Formation

Goal formation in OCP is done via PLN inference. In general, what PLN does for goal formation is to look for predicates that can be proved to probabilistically imply the existing goals. These new predicates will then tend to receive RFS's, according to the logic of RFS's to be outlined later.

As an example of the goal formation process, a simple initial ubergoal (not an adequate one, just a simple initial example) could be

```
AND

    Pleasure

    RecentLearning
```

where Pleasure and RecentLearning are FeelingNodes assessing the amount of ìpleasureî stimulus and the amount of recent learning that the system has undertaken. It may then be learned that whenever the teacher gives it a reward token, this gives it pleasure

```
SimultaneousAttraction

    Evaluation give (teacher, me, reward_token)

    Pleasure
```

This information allows the system (the Goal Formation CIM-Dynamic) to nominate the atom

```
EvaluationLink give (teacher, me, reward_token)
```

as a goal (a subgoal of the original supergoal). This is an example of goal refinement, which is one among many ways that PLN can create new goals from existing ones.

Goal Fulfillment and Predicate Schematization

When there is a Goal Atom G important in the system (with a lot of RFS), the GoalFulfillment CIMDynamic seeks SchemaNodes S that it has reason to believe, if enacted, will cause G to become true (satisfied). It then adds these to the ActiveSchemaPool, an object to be discussed below. The dynamics by which the GoalFulfillment process works will be discussed below in the section on economic goal and action selection.

For example, if a Context Node C has a high truth value at that time (because it is currently satisfied), and is involved in a relation

```
SimultaneousAttraction

    C

    PredictiveAttraction S G
```

(for some SchemaNode S and Goal Node G) then this SchemaNode S is likely to be selected by the GoalFulfillment process for execution. The process may also allow the importance of various schema S to bias its choices of which schemata to execute.

The formation of these schema-context-goal triads may occur according to generic inference mechanisms. However, a specially-focused PredicateSchematization CIMDynamic is very useful here as a mechanism of inference control, increasing the number of such relations that will exist in the system.

Context Formation

New contexts are formed by a combination of processes:

- The MapEncapsulation CIMDynamic, which creates Context Nodes embodying repeated patterns in the perceived world
- The Evolutionary and Logical ConceptCreation MindAgents, which fuse and split Context Nodes to create new ones.
- The Clustering MindAgent which creates ConceptNodes representing sets of similar Context Nodes; aided by the ConceptBasedPredicateFormation MindAgent which then builds Context Nodes corresponding to these ConceptNodes.

Essentially, the map encapsulation process results in the creation of

- Maps joining Context Nodes involving Atoms that have high interaction-channel activation at the same time, with respect to the same interaction channel.
- Maps joining Context Nodes that are involved in a temporal activation pattern that recurs at multiple points in the systemís experience

The details of this process will be presented in the following chapter, where it will be shown how context formation is a special case of the more general process of map encapsulation.

On the other hand the concept creation process results in existing Context Nodes splitting into sub-nodes reflecting subcontexts, or pairs of existing Context Nodes fusing into new (more general, more specific, or simply related) Context Nodes. And clustering, often acting in concert with these other processes, creates more abstract and general Context Nodes out of more specific ones.

Execution Management

The OpenCogPrime:GoalFulfillment CIM-Dynamic chooses schemata that are found likely to achieve current goals, but it doesn't actually execute these schemata. What it does is to take these schemata and place them in a container called the OpenCogPrime:ActiveSchemaPool.

The OpenCogPrime:ActiveSchemaPool contains a set of schemata that have been determined to achieve the current goal-set. I.e., everything in the active schema pool should be a schema S so that it has been concluded

```
PredictiveAttraction <s>

                S
```

G

where G is one of the goals in the current goal pool.

The decision of which schemata in the ActiveSchemaPool to enact is made by an object called the OpenCogPrime:ExecutionManager, which is invoked each time the OpenCogPrime:SchemaActivation CIM-Dynamic is executed. The ExecutionManager is used to select which schemata to execute, based on doing reasoning and consulting memory regarding which active schemata can usefully be executed simultaneously without causing *destructive interference* (and hopefully causing constructive interference). This process will also sometimes (indirectly) cause new schemata to be created and/or other schemata from the AtomTable to be made active. This process is described more fully in the pages on OpenCogPrime:EconomicGoalSelection and OpenCogPrime:EconomicActionSelection.

Goals and Time

The OCP system maintains an explicit list of "ubergoals", which as will be explained in a later section, receive attentional currency which they may then allocate to their subgoals according to a particular mechanism.

It is quite possible to have a OCP system with multiple ubergoals. However, in our work so far we have been assuming a single ubergoal named "Satisfaction," so that the ubergoal is: Maximize Satisfaction, as measured by the Satisfaction FeelingNode.

In our practical work so far, the splitting of the Satisfaction ubergoal into subgoals corresponding to particular subcomponents of Satisfaction is then be achieved

- in part, by the system's learning processes
- in part, via explicit programmer-specification, carried out via defining the Satisfaction FeelingNode's internal evaluator in terms of other complex predicates, such as GainingUnderstanding (a grounded SchemaNode that measures the amount of new learning occurring in the system) or Reward (a FeelingNode corresponding to the receipt of reward from human teachers, e.g. from Reward perceptions coming in from AGISim).

(Domain-specific applications of the OCP system may involve the creati on of application-specific goals, such as creating a particular type of predicate, recognizing patterns in a certain type of biological data, giving humans satisfactory answers to their questions, etc. These may be wired into the basic SatisfactionMaximization goal, and also given a substantial importance on their own.)

All this is very simple conceptually, but it leaves out one very important factor: time. The truth value of a FeelingNode is averaged over the *relevant past,* but the time scale of this averaging can be very important. In very many cases, it may be worthwhile to have separate FeelingNodes measuring exactly the same thing, but doing their truth-value time-averaging over different time scales. In fact this is the case for all the elementary FeelingNodes listed above: InternalNovelty, Health, GainingUnderstanding, ObservedUserSatisfaction etc. Corresponding to each of these FeelingNodes as described above, we may posit a *short-term* version, leading to such things as CurrentInternalNovelty, CurrentHealth, etc. It is possible that even more flexibility than this may be useful, i.e. more than 2 different variants of the same FeelingNode, all with different time scales.

The most interesting case, however, is CurrentSatisfaction. What is interesting is that it may be specifically valuable *not* to have CurrentSatisfaction and Satisfaction constructed the same way. The reason is that, if

CurrentSatisfaction is different from Satisfaction, then there can be a CurrentSatisfactionMaximization goal, which seeks specifically to maximize the qualities that have been associated with CurrentSatisfaction.

Of course, all this time-dependence could be left for the system to figure out all by itself. In figuring out how to best achieve Satisfaction, the system will create short-term goals, and reason that achieving these short-term goals may be the best way to achieve its long-term goals. The building-in of Feelings and Goals with particular temporal structure is an assertion that this time-dependent nature of Satisfaction is an extremely fundamental aspect of mind that perhaps should not have to be learned on the individual-mind level (though obviously in human history and prehistory it was *learned* in the evolutionary sense).

Given a multi-time-scale notion of Satisfaction, one valuable method of system control is for the system to purposefully modify the CurrentSatisfaction FeelingNode. I.e., it may judge that in order for it to fulfill its long-term goals, its short-term goals should be so-and-such, and it may embody *so-and-such* in the CurrentSatisfaction FeelingNode.

An example of a *Satisfaction control schema* of this type would be the following

```
ImplicationLink
    AND
        PredictiveImplicationLink $X CurrentSatisfaction
        PredictiveImplicationLink $X (NOT Satisfaction)
    PredictiveImplicationLink
        $X
        NOT CurrentSatisfaction
```

The rule says: If X makes you happy in the short run, but achieving X is acting against long-term Satisfaction, then revise your CurrentSatisfaction so that X no longer makes you happy in the short run. Needless to say, humans do not have this kind of explicit control over the drivers of their short-time-scale Satisfaction; but there is no reason that a OCP system cannot. This kind of self-modification does not require sourcecode modification, but *merely* the learning of relatively compact (but fairly abstract) cognitive schemata.

Economic Goal and Action Selection

Now we turn to the topic of goal and action selection — the algorithmically subtlest issue addressed in this chapter, which is addressed in OCP via an extension of the artificial-economics approach to attention allocation, discussed above. The main actors (apart from the usual ones like the AtomTable, economic attention allocation, etc.) in the tale to be told here are as follows:

- Structures:
  - OpenCogPrime:UbergoalPool
  - OpenCogPrime:ActiveSchemaPool
- MindAgents:
  - OpenCogPrime:GoalBasedSchemaSelection
  - OpenCogPrime:GoalBasedSchemaLearning
  - OpenCogPrime:GoalAttentionAllocation
  - OpenCogPrime:FeasibilityUpdating
  - OpenCogPrime:SchemaActivation

Ubergoal Pool

The Ubergoal Pool contains the Atoms that the system considers as top-level goals. These goals must be treated specially by attention allocation: they must be given funding by the Unit so that they can use it to pay for getting themselves achieved. The weighting among different top-level goals is achieved via giving them differential amounts of currency. STICurrency is the key kind here, but of course ubergoals must also get some LTICurrency so they won't be forgotten. (Inadvertently deleting your top-level supergoals from memory is generally considered to be a bad thing ... it's in a sense a sort of suicide...)

Transfer of STI "Requests for Service" Between Goals

Transfer of 'attentional funds' from goals to subgoals, and schema modules to other schema modules in the same schema, takea place via a mechanism of promises of funding (or 'requests for service,' to be called 'RFS's' from here on). This mechanism relies upon and interacts with ordinary economic attention allocation but also has special properties.

The logic of these RFS's is as follows. If agent A issues a RFS of value x to agent B, then

1. When B judges it appropriate, B may redeem the note and ask A to transfer currency of value x to B.
2. A may withdraw the note from B at any time.

(There is also a little more complexity here, in that we will shortly introduce the notion of RFS's whose value is defined by a set of constraints. But this complexity does not contradict the two above points.) The total value of the of RFS's possessed by an Atom may be referred to as its 'promise.'

Now we explain how RFS's may be passed between goals. Given two predicates A and B, if A is being considered as a goal, then B may be considered as a subgoal of A (and A the supergoal of B) if there exists a Link of the form

```
PredictiveImplication B A
```

I.e., achieving B may help to achieve A. Of course, the strength of this link and the temporal characteristics of this link are important in terms of quantifying how strongly and how usefully B is a subgoal of A.

Supergoals (not only top-level ones, aka ubergoals) allocate RFS's to subgoals as follows. Supergoal A may issue a RFS to subgoal B if it is judged that achievement (i.e., predicate satisfaction) of B implies achievement of A. This may proceed recursively: subgoals may allocate RFS's to subsubgoals according to the same justification.

Unlike actual currency, RFS's are not conserved. However, the actual payment of real currency upon redemption of RFS's obeys the conservation of real currency. This means that agents need to be responsible in issuing and withdrawing RFS's. In practice this may be ensured by having agents follow a couple simple rules in this regard.

1. If B and C are two alternatives for achieving A, and A has x units of currency, then A may promise both B and C x units of currency. Whomever asks for a redemption of the promise first, will get the money, and then the promise will be rescinded from the other one.

2. On the other hand, if the achievement of A requires both B and C to be achieved, then B and C may be granted RFS's that are defined by constraints. If A has x units of currency, then B and C receive an RFS tagged with the constraint (B+C<x). This means that in order to redeem the note, either one of B or C must confer with the other one, so that they can simultaneously request constraint-consistent amounts of money from A.

As an example of the role of constraints, consider the goal of playing fetch successfully (a subgoal of "get reward").... Then suppose it is learned that

```
ImplicationLink

     AND

          get_ball

          deliver_ball

     play_fetch
```

Then, if play_fetch has $10 in STICurrency, it may know it has $10 to spend on a combination of get_ball and deliver_ball. In this case both get_ball and deliver_ball would be given RFS's labeled with the contraint

```
RFS.get_ball + RFS.deliver_ball <= 10
```

The issuance of RFS's embodying constraints is different from (and generally carried out prior to) the evaluation of whether the constraints can be fulfilled.

An ubergoal may rescind offers of reward for service at any time. And, generally, if a subgoal gets achieved and has not spent all the money it needed, the supergoal will not offer any more funding to the subgoal (until/unless it needs that subgoal achieved again).

As there are no ultimate sources of RFS in OCP besides ubergoals, promise may be considered as a measure of 'goal-related importance.'

Transfer of RFS's among Atoms is carried out by the GoalAttentionAllocation MindAgent.

Feasibility Structures

Next, there is a numerical data structure associated with goal Atoms, which is called the feasibility structure. The feasibility structure of an Atom G indicates the feasibility of achieving G as a goal using various amounts of effort. It contains triples of the form (t,p,E) indicating the truth value t of achieving goal G to degree p using effort E. Feasibility structures must be updated periodically, via scanning the links coming into an Atom G; this may be done by a FeasibilityUpdating MindAgent. Feasibility may be calculated for any Atom G for which there are links of the form

```
Implication

     Execution S

     G
```

for some S. Once a schema has actually been executed on various inputs, its cost of execution on other inputs may be empirically estimated. But this is not the only case in which feasibility may be estimated. For example, if goal G inherits from goal G1,and most children of G1 are achievable with a certain feasibility, then probably G is achievable with that same feasibility as well. This allows feasibility estimation even in cases where no plan for achieving G yet exists, e.g. if the plan can be produced via predicate schematization, but such schematization has not yet been carried out.

Feasibility then connects with importance as follows. Important goals will get more STICurrency to spend, thus will be able to spawn more costly schemata. So, the GoalBasedSchemaSelection MindAgent, when choosing which schemata to push into the ActiveSchemaPool, will be able to choose more costly schemata corresponding to goals with more STICurrency to spend.

Goal Based Schema Selection

Next, the GoalBasedSchemaSelection selects schemata to be placed into the ActiveSchemaPool. It does this by choosing goals G, and then choosing schemata that are alleged to be useful for achieving these goals. It chooses goals via a fitness function that combines promise and feasibility. This involves solving an optimization problem: figuring out how to maximize the odds of getting a lot of goal-important stuff done within the available amount of (memory and space) effort. Potentially this optimization problem can get quite subtle, but initially some simple heuristics are satisfactory. (One subtlety involves handling dependencies between goals, as represented by constraint-bearing RFS's.)

Given a goal, the GBSS MindAgent chooses a schema to achieve that goal via the heuristic of selecting the one that maximizes a fitness function balancing the estimated effort required to achieve the goal via executing the schema, with the estimated probability that executing the schema will cause the goal to be achieved.

When searching for schemata to achieve G, and estimating their effort, one factor to be taken into account is the set of schemata already in the ActiveSchemaPool. Some schemata S may simultaneously achieve two goals; or two schemata achieving different goals may have significant overlap of modules. In this case G may be able to get achieved using very little or no effort (no additional effort, if there is already a schema S in the ActiveSchemaPool that is going to cause G to be achieved). But if G decides it can be achieved via a schema S already in the ActiveSchemaPool, then it should still notify the ActiveSchemaPool of this, so that G can be added to S's index (see below). If the other goal G1 that placed S in the ActiveSchemaPool decides to withdraw S, then S may need to hit up G1 for money, in order to keep itself in the ActiveSchemaPool with enough funds to actually execute.

SchemaActivation

And what happens with schemata that are actually in the ActiveSchemaPool? Let us assume that each of these schema is a collection of modules, connected via ActivationLinks, which have semantics: (ActivationLink A B)

means that if the schema that placed module A in the schema pool is to be completed, then after A is activated, B should be activated. (We will have more to say about schemata, and their modularization, in the following chapter.)

When a goal places a schema in the ActiveSchemaPool, it grants that schema an RFS equal in value to the (some fraction of) the (promissory+real) currency it has in its possession. The heuristics for determining how much currency to grant may become sophisticated; but initially we may just have a goal give a schema all its promissory currency; or in the case of a top-level supergoal, all its actual currency.

When a module within a schema actually executes, then it must redeem some of its promissory currency to turn it into actual currency, because executing costs money (paid to the Lobe). Once a schema is done executing, if it hasn't redeemed all its promissory currency, it gives the remainder back to the goal that placed it in the ActiveSchemaPool.

When a module finishes executing, it passes promissory currency to the other modules to which it points with ActivationLinks.

The network of modules in the ActiveSchemaPool is a digraph (whose links are ActivationLinks), because some modules may be shared within different overall schemata. Each module must be indexed via which schemata contain it, and each schema must be indexed via which goal(s) want it in the ActiveSchemaPool.

GoalBasedSchemaLearning

Finally, we have the process of trying to figure out how to achieve goals, i.e. trying to learn links between ExecutionLinks and goals G. This process should be focused on goals that have a high importance but for which feasible achievement-methodologies are not yet known. Predicate schematization is one way of achieving this; another is MOSES procedure evolution.

Context-Schema-Goal Triads

A particularly critical aspect of intelligence, we propose, is the *Context-Schema-Goal triad*: a context, schema and goal that are interlinked according to the general pattern

enactment of the schema S in the context C is likely to result in the attainment of the goal state G.

Technically, this means:

```
Context

    C

    PredictiveAttraction

        Evaluation done S

        G
```

(here done is a built-in schema which basically indicates that S has been executed on some input), or

```
Context

    C

    PredictiveAttraction

        Execution S A *

        G
```

where A is some particular input.

In OCP, sometimes a context-schema-goal triad will be represented on the Atom level, sometimes on the map level. These triads form the essence of OCP's *behavioral reinforcement learning* subsystem, which allows the system to learn new behaviors through experience. These *behaviors* include cognitive behaviors — i.e. the learning of the *cognitive schemata* that have been discussed frequently in previous chapters.

Feeling Nodes

The general concept of a FeelingNode is that of an *internal sensor* — a FeelingNode is a mechanism that a OCP system uses to sense some aspect of itself. FeelingNodes are not intended to represent emotions, which we understand as complex distributed phenomena, not encapsulable in single nodes. There are some parallels and relationships between FeelingNodes and emotions, which will arise sometimes in our discussion of FeelingNodes, but we don't intend these parallels to be taken very seriously. The study of emotions in AGI systems will be a subtle science, we believe, but one that will be difficult to seriously explore prior to the development of powerful AGI systems that can communicate fluently with humans.

Like Goal and Context Atoms, FeelingNodes, structurally, are simply PredicateNodes. Specifically, they are a subclass of OpenCogprime:GroundedPredicateNode), which is grounded in a Procedure that measures either

- some sort of global indicator of the state of the system, or
- an in-built feeling indicator, such as the Pleasure FeelingNode, which is activated via things such as getting rewarded by the teacher

More technically, they are PredicateNodes whose internal schema expression trees have at their leaves (along with, perhaps, other inputs) either

- FeelingSchemata, or
- OutputValues of FeelingNodes

A set of elementary FeelingSchemata is defined; some FeelingNodes may simply wrap up a single FeelingSchema. Others, containing combinations of elementary FeelingSchema are called compound FeelingNodes.

A FeelingNode's truth value will vary over time, in accordance with the variance of the output of the elementary FeelingSchemata it's defined in terms of.

Learning new FeelingSchemata and modifying or deleting existing ones are advanced system dynamics, which may be undertaken only by advanced OCP systems that are able to learn procedures based on complex experience-based inferences.

Note that the importance of a FeelingNode bears no direct relationship to its truth value.

For instance, the *Satisfaction FeelingNode* (representing the degree to which the system's goals are satisfied, overall) may have a high importance but a low truth value — this might (very loosely speaking) be considered a variety of *depression.* Also, FeelingNodes may be embedded in Goal Atoms, with either positive or negative valence. One could have a goal of being satisfied (having a high truth value for the Satisfaction FeelingNode), a goal of being not overworked (having a low truth value for the SystemStress FeelingNode), etc.

The key thing distinguishing elementary feelings from compound feelings is that elementary feelings are evaluable completely automatically, without any thought or any adaptation based on experience. From the experiencing OCP mind's point of view, these feelings simply are what they are. Only once OCP reaches the point of analyzing and modifying its source code, will it have control over its elementary feelings.

However, any OCP may have *control* over the compound feelings it generates. This leads to some interesting philosophical and psychological issues. The notion of *control* becomes bound up with the Consciousness distinction and the problem of will. When we say a OCP has control over the compound FeelingNodes it generates and utilizes, we mean that its mental processes may adapt these based on its experience, either in service of its goals or spontaneously. Whether it feels subjectively that it has control over these FeelingNodes and associated dynamics, is a different question entirely — which gets into the complex issue of the nature of emotions in an non-evolved, nonhuman intelligence. One thing that can potentially happen is that FeelingNodes are extensively created and modified and utilized via non-goal-oriented self-organizing dynamics, and goal-satisfaction-oriented dynamics fail to exert significant influence over them. In this case, the system could rationally come to model itself as being unable to achieve its goals because its feelings were *running amok.*

A Provisional List of Elementary Feelings

Here we will define a set of initial, elementary feelings for OCP. A set of initial, elementary goals will follow from these. The initial feeling set described here is the one we used for initial experimentation in prototypes, but it's very much a *first draft* — we envision that further work will lead to substantial modifications and additions.

TeacherSatisfaction — How happy are the OCP instance's human friends with what the system is doing?

This may be gauged in several possible ways depending on the OCP front end in use:

- explicit user responses to system outputs (e.g., the user clicking a reward button, in the AGISim user interface)
- interpretation of the user's natural language responses (*Good boy, OCP!*)
- objective measures of user response: how long does the user view the results returned by a OCP-powered search engine, etc.

Gaining Understanding — How much new pattern is being formed in the system overall?

One crude way to assess this is: what is the total s*d value of set new atoms, normalized by the total size of the atoms in this set? (The normalization by size is only needed because of the existence of atoms containing compound schema and predicate expressions.) This is an important system goal: the system is supposed to want to learn things!

External Novelty — How much is the perceived world of the system changing over time?

A love for novelty encourages exploration which broadens the mind.

Internal Novelty — How much is the internal landscape changing over time, the Atomspace itself?

Health — a combination of a number of elementary indicators to give a composite indicator of health.

These indicators may include such things as free memory and response time for certain types of queries submitted by users. One may also create individual FeelingSchema for these elementary indicators, such as FreeSystemMemory or QueryResponseTime (in a OCP-driven query-oriented software application), etc. — a strategy which allows OCP to construct its own composite Health feelings.

Satisfaction — a compound feeling which is a combination (for example, a weighted average) of the feelings mentioned above.

This is the most critical FeelingNode, as it is the basis of OCP's motivational system. Of course, the use of the English word *Satisfaction* is not entirely accurate here; but we've found that any term with any evocative power also has the power to mislead, and after rejecting *Happiness* as being too simplistically misleading, we've settled on *Satisfaction.*

What makes *Satisfaction* OCP's primary motivator is simply the fact that the Goal Node referring to the goal of having high Satisfaction (i.e. of the FeelingNode having a high truth value) is constantly *autonomically* given a high Short-Term Importance. This causes a large percentage of the processes of the system to focus on the Satisfaction FeelingNode in various direct and indirect ways.

This initial feeling framework is intentionally very simplistic. The intention is that more complex feelings should emerge via the natural evolutionary processes of the system. Imposing a subtle system of human-like feelings on OCP is probably not desirable. Rather, we should give it a very simple framework for sensing itself, and allow higher-level emotions to emerge as is appropriate for the system given its particular goals and contexts.

Predictive Implication and Attraction

This page briefly reviews the notions of predictive implication and predictive attraction, which are critical to many aspects of OCP dynamics including goal-oriented behavior.

Define

```
Attraction A B <s>
```
as P(B||A) ñ P(B||~A) = s, or in node and link terms

```
s = (Inheritance A B).s ñ (Inheritance ~A B).s
```

For instance

```
Attraction fat pig.s = (Inheritance fat pig).s ñ (Inheritance ~fat pig).s
```

Relatedly, in the temporal domain, we have the link type PredictiveImplication, where

```
PredictiveImplication A B <s>
```

roughly means that s is the probability that

```
Implication A B <s>
```

holds and also A occurs before B. More sophisticated versions of PredictiveImplication come along with more specific information regarding the time lag between A and B: for instance a time interval T in which the lag must lie, or a probability distribution governing the lag between the two events.

We may then introduce

```
PredictiveAttraction A B <s>
```

to mean

```
s = (PredictiveImplication A B).s ñ (PredictiveImplication ~A B).s
```

For instance

```
(PredictiveAttraction kill_Ben be_happy.s = (PredictiveImplication kill_Ben be_happy).s ñ
(PredictiveImplication ~kill_Ben be_happy).s
```

This is what really matters in terms of determining whether killing Ben is worth doing in pursuit of the goal of being happy ñ not just how likely it is to be happy if you kill Ben, but how differentially likely it is to be happy if you kill Ben.

Along with predictive implication and attraction, sequential logical operations are important, represented by operators such as SequentialAND, SimultaneousAND and SimultaneousOR. For instance

```
PredictiveAttraction

    SequentialAND

        Teacher says ìfetchî

        I get the ball

        I bring the ball to the teacher
```

```
I get a reward
```

combines SequentialAND and PredictiveAttraction. In this manner, an arbitrarily complex system of serial and parallel temporal events can be constructed.

Chapter Eleven: Procedure Execution

Procedure Execution

- OpenCogPrime:ProcedureExecutionDetails

In this brief chapter we delve into the details of the Combo tree evaluation process. This is somewhat of a "mechanical," implementation-level topic, yet it also involves some basic conceptual points, on which OCP as an AGI design does procedure evaluation fundamentally differently from narrow-AI systems or conventional programming language interpreters. Basically, what makes OCP Combo tree evaluation somewhat subtle due to the interfacing between the Combo evaluator itself and the rest of the OCP system.

In the current OCP design, Procedure objects (which contain Combo trees, and are associated with ProcedureNodes) are evaluated by ProcedureEvaluator objects. Different ProcedureEvaluator objects may evaluate the same Combo tree in different ways.

In this chapter we will mention three different ProcedureEvaluators: the

- OpenCogPrime:SimpleComboTreeEvaluator,
- OpenCogPrime:EffortBasedComboTreeEvaluator, which is more complex but is required for integration of inference with procedure evaluation (a prototype of this approach was developed within the Novamente Cognition Engine, and porting to OCP would be conceptually straightforward)
- AdaptiveEvaluationOrderComboTreeEvaluator, which is necessary for robust general intelligence but has not been implemented nor designed in detail

In the following section we will delve more thoroughly into the interactions between inference and procedure evaluation.

Another issue related to procedure execution, which rose to the fore in the previous chapter, is the modularization of procedures. This issue however is actually orthogonal to the distinction between the three ProcedureEvaluators mentioned above. Modularity simply requires that particular nodes within a Combo tree be marked as "module roots", so that they may be extracted from the Combo tree as a whole and treated as separate modules, if the ExecutionManager judges this appropriate.

OpenCogPrime:SimpleComboTreeEvaluator

The SimpleComboTreeEvaluator simply does Combo tree evaluation as described earlier. When an NMAtom is encountered, it looks into the AtomTable to evaluate the object.

In the case that an NMSchema refers to an ungrounded SchemaNode, and an appropriate EvaluationLink value isn't in the AtomTable, there's an evaluation failure, and the whole procedure evaluation returns the truth value (.5,0): i.e., a zero-weight-of-evidence truth value, which is equivalent essentially to returning no value.

In the case that an NMPredicate refers to an ungrounded PredicateNode, and an appropriate EvaluationLink isn't in the AtomTable, then some very simple "default thinking" is done, and it is assigned the truth value of the predicate on the given arguments to be the TruthValue of the corresponding PredicateNode (which is defined as the mean truth value of the predicate across all arguments known to OCP)

OpenCogPrime:EffortIntervalComboTreeEvaluator

The next step is to introduce the notion of 'effort' ' the amount of effort that the OCP system must undertake in order to carry out a procedure evaluation. The notion of effort is encapsulated in Effort objects, which may take various forms. The simplest Effort objects measure only elapsed processing time; more advanced Effort objects take into consideration other factors such as memory usage.

An effort-based Combo tree evaluator keeps a running total of the effort used in evaluating the Combo tree. This is necessary if inference is to be used to evaluate NMPredicates, NMSchema, NMArguments, etc. Without some control of effort expenditure, the system could do an arbitrarily large amount of inference to evaluate a single NMAtom.

The matter of evaluation effort is nontrivial because in many cases a given node of a Combo tree may be evaluated in more than one way, with a significant effort differential between the different methodologies. If a Combo tree Node refers to a predicate or schema that is very costly to evaluate, then the ProcedureEvaluator managing the evaluation of the Combo tree must decide whether to evaluate it directly (expensive) or estimate the result using inference (cheaper but less accurate). This decision depends on how much effort the ProcedureEvaluator has to play with, and what percentage of this effort it finds judicious toapply to the particular Combo tree Node in question.

In the relevant prototypes we built within the NCE, this kind of decision was made based on some simple heuristics inside ProcedureEvaluator objects. However, it's clear that, in general, more powerful intelligence must be applied here, so that one needs to have ProcedureEvaluators that ' in cases of sub-procedures that are both important and highly expensive — use PLN inference to figure out how much effort to assign to a given subproblem.

The simplest useful kind of effort-based Combo tree evaluator is the EffortIntervalComboTreeEvaluator, which utilizes an Effort object that contains three numbers (yes, no, max). The yes parameter tells it how much effort should be expended to evaluate an NMAtom if there is a ready answer in the AtomTable. The no parameter tells it how much effort should be expended in the case that there is not a ready answer in the AtomTable. The max parameter tells it how much effort should be expended, at maximum, to evaluate all the NMAtoms in the Combo tree, before giving up. Zero effort is defined as simply looking into the AtomTable.

Quantification of amounts of effort is nontrivial, but a simple heuristic guideline is to assign one unit of effort for each inference step. Thus, for instance,

- (yes, no, max) = (0,5,1000) means that if an NMAtom can be evaluated by AtomTable lookup, this is done, but if AtomTable lookup fails, a minimum of 5 inference steps are done to try to do the evaluation. It also says that no more than 1000 evaluations will be done in the course of evaluating the Combo tree
- (yes, no, max) = (3,5,1000) says the same thing, but with the change that even if evaluation could be done by direct AtomTable lookup, 3 inference steps are tried anyway, to try to improve the quality of the evaluation

OpenCogPrime:AdaptiveEvaluationOrderComboTreeEvaluator

While tracking effort enables the practical use of inference within Combo tree evaluation, if one has truly complex Combo trees, then a higher degree of intelligence is necessary to guide the evaluation process appropriately. The order of evaluation of a Combo tree may be determined adaptively, based on up to three things:

- The history of evaluation of the Combo tree
- Past history of evaluation of other Combo tree's, as stored in a special AtomTable consisting only of relationships about Combo tree-evaluation-order probabilities
- New information entering into OCP's primary AtomTable during the course of evaluation

ProcedureEvaluator objects may be selected at runtime by cognitive schemata, and they may also utilize schemata and MindAgents internally. The AdaptiveEvaluationOrderComboTreeEvaluator is more complex than the other ProcedureEvaluators discussed, and will involve various calls to OCP MindAgents, particularly those concerned with PLN inference.

The Procedure Evaluation Process

Now we give a more thorough treatment of the procedure evaluation process, as embodied in the effort-based or adaptive-evaluation-order evaluators discussed above. The process of procedure evaluation is somewhat complex, because it encompasses three interdependent processes:

- The mechanics of procedure evaluation, which in the OCP design involves traversing Combo trees in an appropriate order. When a Combo tree node referring to a predicate or schema is encountered during Combo tree traversal, the process of predicate evaluation or schema execution must be invoked.
- The evaluation of the truth values of predicates — which involves a combination of inference and (in the case of grounded predicates) procedure evaluation
- The computation of the truth values of schemata — which may involve inference as well as procedure evaluation

We now review each of these processes.

Truth Value Evaluation

Now, what happens when the procedure evaluation process encounters a Combo tree Node that represents a predicate or compound term? The same thing as when some other OCP process decides it wants to evaluate the truth value of a PredicateNode or CompoundTermNode: the generic process of predicate evaluation is initiated.

This process is carried out by a TruthValueEvaluator object. There are several varieties of TruthValueEvaluator, which fall into the following hierarchy:

```
TruthValueEvaluator

    DirectTruthValueEvaluator (abstract)

        SimpleDirectTruthValueEvaluator

    InferentialTruthValueEvaluator (abstract)

        SimpleInferentialTruthValueEvaluator

    MixedTruthValueEvaluator
```

A DirectTruthValueEvaluator evaluates a grounded predicate by directly executing it on one or more inputs; an InferentialTruthValueEvaluator evaluates via inference based on the previously recorded, or specifically elicited, behaviors of other related predicates or compound terms. A MixedTruthValueEvaluator contains references to a DirectTruthValueEvaluator and an InferentialTruthValueEvaluator, and contains a weight that tells it how to balance the outputs from the two.

Direct truth value evaluation has two cases. In one case, there is a given argument for the predicate; then, one simply plugs this argument in to the predicate's internal Combo tree, and passes the problem off to an appropriately selected ProcedureEvaluator. In the other case, there is no given argument, and one is looking for the truth value of the predicate *in general.* In this latter case, some estimation is required. It is not plausible to evaluate the truth value of a predicate on every possible argument, so one must sample a bunch of arguments and then record the resulting probability distribution. A greater or fewer number of samples may be taken, based on the amount of effort that's been allocated to the evaluation process. It's also possible to evaluate the truth value of a predicate in a given context (information that's recorded via embedding in a ContextLink); in this case, the random sampling is restricted to inputs that lie within the specified context.

On the other hand, the job of an InferentialTruthValueEvaluator is to use inference rather than direct evaluation to guess the truth value of a predicate (sometimes on a particular argument, sometimes in general). There are several different control strategies that may be applied here, depending on the amount of effort allocated. The simplest strategy is to rely on analogy, simply searching for similar predicates and using their truth values as guidance. (In the case where a specific argument is given, one searches for similar predicates that have been evaluated on similar arguments.) If more effort is available, then a more sophisticated strategy may be taken. Generally, an InferentialTruthValueEvaluator may invoke a SchemaNode that embodies an inference control strategy for guiding the truth value estimation process. These SchemaNodes may then be learned like any others.

Finally, a MixedTruthValueEvaluator operates by consulting a DirectTruthValueEvaluator and/or an InferentialTruthValueEvaluator as necessary, and merging the results. Specifically, in the case of an ungrounded PredicateNode, it simply returns the output of the InferentialTruthValueEvaluator it has chosen. But in the case of a GroundedPredicateNode, it returns a weighted average of the directly evaluated and inferred values, where the weight is a parameter. In general, this weighting may be done by a SchemaNode that is selected by the MixedTruthValueEvaluator; and these schemata may be adaptively learned.

Schema Execution

Finally, schema execution is handled similarly to truth value evaluation, but it's a bit simpler in the details. Schemata have their outputs evaluated by SchemaExecutor objects, which in turn invoke ProcedureEvaluator objects. We have the hierarchy:

```
SchemaExecutor

    DirectSchemaExecutor (abstract)

        SimpleDirectSchemaExecutor

    InferentialSchemaExecutor (abstract)

        SimpleInferentialSchemaExecutor

     MixedSchemaExecutor
```

A DirectSchemaExecutor evaluates the output of a schema by directly excecuting it on some inputs; an InferentialSchemaExecutor evaluates via inference based on the previously recorded, or specifically elicited, behaviors of other related schemata. A MixedSchemaExecutor contains references to a DirectSchemaExecutor and an InferentialSchemaExecutor, and contains a weight that tells it how to balance the outputs from the two (not always obvious, depending on the output type in question).

Contexts may be used in schema execution, but they're used only indirectly, via being passed to TruthValueEvaluators used for evaluating truth values of PredicateNodes or CompoundTermNodes that occur internally in schemata being executed.

Chapter Twelve: Dimensional Embedding

Dimensional Embedding

This page and those over which it supervenes

- OpenCogPrime:HarelKorenEmbeddingAlgorithm
- OpenCogPrime:EmbeddingBasedInference

discuss a particular mathematical mechanism that is useful for several purposes: the embedding of Atoms in the OpenCog Atomspace into n-dimensional space.

In general SMEPH terms, this provides a tool for representational transformation of SMEPH hypergraphs into sets of points in dimensional space. For most purposes, the hypergraph representation is more insightful, but there are also sometimes advantages to a dimensional representation, due to the massive body of sophisticated mathematical and computational mechanisms associated with numerical vectors.

In practical OCP terms, this embedding has applications to PLN inference control, and to the guidance of instance generation in PEL learning of Combo trees. It is also, in itself, a valuable and interesting heuristic for

sculpting the link topology of a OCP Atomspace. The basic dimensional embedding algorithm described here is not original to OCP, but it has not previously been applied in these sorts of context.

The role of dimensional embedding in OCP is interesting conceptually as well as technically. The human brain's neural net is embedded in 3D space in a way that OCP's is not. In many ways, the unconstrained geometry of OCP's Atom network is superior. But there are some cases (e.g. PLN control, and PEL guidance) where dimensional geometry provides a useful heuristic for constraining a huge search space, via providing a compact way of storing a large amount of information. Dimensionally embedding Atoms lets OCP be dimensional like the brain when it needs to be, yet with the freedom of nondimensionality the rest of the time. This dual strategy is one that may be of value for AGI generally beyond the OCP design.

There is an obvious way to project OCP or SMEPH Atoms into n-dimensional space, by assigning each Atom a numerical vector based on the weights of its links. But this is not a terribly practical observation, because the vectors obtained in this way will live, potentially, in millions- or billions-dimensional space. What we are doing here is a bit different. We are defining more specific embeddings, each one based on a particular link type or set of link types. And we are doing the embedding into a space whose dimensionality is *high but not too high*, e.g. n=50.

The philosophy underlying the ideas proposed here is similar to that underlying Principal Components Analysis (PCA) in statistics (Joliffe, 2002). The n-dimensional spaces we define here, like those used in PCA or LSI, are defined by sets of *orthogonal concepts* extracted from the original space of concepts. The difference is that PCA and LSI work on spaces of entities defined by feature vectors, whereas the methods described here work for entities defined as nodes in weighted graphs. There is no precise notion of *orthogonality* for nodes in a weighted graph, but one can introduce a reasonable proxy.

Link Based Dimensional Embedding

In this section we define the type of dimensional embedding that we will be talking about. For concreteness we will speak in terms of OCP nodes and links but the discussion applies equally well to SMEPH vertices and edges, and in fact even more generally than that.

A *link based dimensional embedding* is defined as a mapping that maps a set of OCP Atoms into points in an n-dimensional real space, by

- mapping link strength into coordinate values in an embedding space, and
- representing nodes as points in this embedding space, using the coordinate values defined by the strengths of their links.

In the usual case, a dimensional embedding is formed from links of a single type, or from links whose types are very closely related (e.g. from all symmetrical logical links).

Mapping all the link strengths of the links of a given type into coordinate values in a dimensional space is a simple, but not a very effective strategy. The approach described here is based on strategically choosing a subset of particular links and forming coordinate values from them. The choice of links is based on the desire for a correspondence between the metric structure of the embedding space, and the metric structure implicit in the weights of the links of the type being embedded.

More formally, let proj(A) denote the point in R^n corresponding to the Atom A. Then if, for example, we are doing an embedding based on SimilarityLinks, we want there to be a strong correlation between

```
(SimilarityLink A B).tv.s
```

and

```
d_E( proj(A), proj(B) )
```

where d_E denotes the Euclidean distance on the embedding space. This is a simple case because SimilarityLink is symmetric. Dealing with asymmetric links like InheritanceLinks is a little subtler, and will be done below in the context of inference control.

Larger dimensions generally allow greater correlation, but add complexity. If one chooses the dimensionality equal to the number of nodes in the graph, there is really no point in doing the embedding. On the other hand, if one tries to project a huge and complex graph into 1 or 2 dimensions, one is bound to lose a lot of important structure. The optimally useful embedding will be into a space whose dimension is *large but not too large.*

For internal OCP inference purposes, we should generally use a moderately high-dimensional embedding space, say n=50 or n=100.

Harel and Koren's Dimensional Embedding Algorithm

Our technique for embedding OCP Atoms into high-dimensional space is based on an algorithm suggested by David Harel and Yehuda Koren (XX insert reference). Their work is concerned with visualizing large graphs, and they propose a two-phase approach:

1. embed the graph into a high-dimensional real space
2. project the high-dimensional points into 2D or 3D space for visualization

In OCP, we don't always require the projection step (step 2); our focus is on the initial embedding step. Harel and Koren's algorithm for dimensional embedding (step 1) is directly applicable to the OCP context.

Of course this is not the only embedding algorithm that would be reasonable to use in an OCP context; it's just one possibility that seems to make sense.

Their algorithm works as follows.

Suppose one has a graph with symmetric weighted links. Further, assume that between any two nodes in the graph, there is a way to compute the weight that a link between those two nodes would have, even if the graph in fact doesn't contain a link between the two nodes.

In the OCP context, for instance, the nodes of the graph may be ConceptNodes, and the links may be SimilarityLinks. We will discuss the extension of the approach to deal with asymmetric links like InheritanceLinks, later on.

Let n denote the dimension of the embedding space (e.g. n=50). We wish to map graph nodes into points in Rn, in such a way that the weight of the graph link between A and B correlates with the distance between proj(A) and proj(B) in Rn.

Step 1)

Choose n "pivot points" that are roughly uniformly distributed across the graph.

To do this, one chooses the first pivot point at random and then iteratively chooses the i'th point to be maximally distant from the previous (i-1) points chosen.

One may also use additional criteria to govern the selection of pivot points. In OCP, for instance, we may use *long-term stability* as a secondary criterion for selecting Atoms to serve as pivot points. Greater computational efficiency is achieved if the pivot-point Atoms don't change frequently.

Step 2)

Estimate the similarity between each Atom being projected, and the n pivot Atoms

Step 2 is expensive. However, the cost is decreased somewhat in the OCP case by caching the similarity values produced in a special table (they may not be important enough otherwise to be preserved in OCP). Then, in cases where neither the pivot Atom nor the Atom being compared to it have changed recently, the cached value can be reused.

Step 3)

Create an n-dimensional space by assigning a coordinate axis to each pivot Atom. Then, for an Atom i, the i'th coordinate value is given by its similarity to the i'th pivot Atom

After Step 3, one has transformed one's graph into a collection of n-dimensional vectors

Embedding Based Inference Control

One important application for dimensional embedding in OCP is to help with the control of

- Logical inference
- Direct evaluation of logical links

We describe how it can be used specifically to stop the OCP system from continually trying to make the same unproductive inferences.

To understand the problem being addressed, suppose the system tries to evaluate the strength of the relationship

```
SimilarityLink foot toilet
```

Assume that no link exists in the system representing this relationship.

Here "foot" and "toilet" are hypothetical ConceptNodes that represent aspects of the concepts of foot and toilet respectively. In reality these concepts might well be represented by complex maps rather than individual nodes.

Suppose the system determines that the strength of this Link is very close to zero. Then (depending on a threshold in the MindAgent), it will probably not create a SimilarityLink between the "foot" and "toilet" nodes.

Now, suppose that a few cycles later, the system again tries to evaluate the strength of the same Link,

```
SimilarityLink foot toilet
```

Again, very likely, it will find a low strength and not create the Link at all.

The same problem may occur with InheritanceLinks, or any other (first or higher order) logical link type.

Why would the system try, over and over again, to evaluate the strength of the same nonexistent relationship? Because the control strategies of the inference and logical relationship mining MindAgents are simple. These MindAgents work by selecting Atoms from the AtomTable with probability proportional to importance, and trying to build links between them. If the foot and toilet nodes are both important at the same time, then these MindAgents will try to build links between them — regardless of how many times they've tried to build links between these two nodes in the past and failed.

How do we solve this problem using dimensional embedding? Generally:

- one will need a different embedding space for each link type for which one wants to prevent repeated attempted inference of useless relationships [though, sometimes, very closely related link types might share the same embedding space; this must be decided on a case-by-case basis]
- in the embedding space for a link type L, one obviously only wants to embed Atoms of a type that can be related by links of type L

It is too expensive to create a new embedding very often. Fortunately, when a new Atom is created or an old Atom is significantly modified, it's easy to reposition the Atom in the embedding space by computing its relationship to the pivot Atoms. Once enough change has happened, however, new pivot Atoms will need to be recomputed, which is a substantial computational expense. We must update the pivot point set every N cycles, where N is large; or else, whenever the total amount of change in the system has exceeded a certain threshold.

Now, how is this embedding used for inference control? Let's consider the case of similarity first. Quite simply, one selects a pair of Atoms (A,B) for SimilarityMining (or inference of a SimilarityLink) based on some criterion such as:

```
importance(A)*importance(B) * simproj(A,B)
```

where

```
distproj(A,B) = dE( proj(A) , proj(B) )

simproj = 2-c*distproj
```

and c is an important tunable parameter.

What this means is that, if A and B are far apart in the SimilarityLink embedding space, the system is unlikely to try to assess their similarity.

There is a tremendous space efficiency of this approach, in that, where there are N Atoms and m pivot Atoms, N^2 similarity relationships are being approximately stored in m*N coordinate values. Furthermore, the cost of computation is m*N times the cost of assessing a single SimilarityLink. By accepting crude approximations of actual similarity values, one gets away with linear time and space cost.

Because this is just an approximation technique, there are definitely going to be cases where A and B are not similar, even though they're close together in the embedding space. When such a case is found, it may be useful for the Atomspace to explicitly contain a low-strength SimilarityLink between A and B. This link will prevent the system from making false embedding-based decisions to explore (SimilarityLink A B) in the future. Putting explicit low-strength SimilarityLinks in the system in these cases, is obviously much cheaper than using them for all cases.

We've been talking about SimilarityLinks, but the approach is more broadly applicable. Any symmetric link type can be dealt with about the same way. For instance, it might be useful to keep dimensional embedding maps for

```
SimilarityLink
ExtensionalSimilarityLink
EquivalenceLink
ExtensionalEquivalenceLink
```

On the other hand, dealing with asymmetric links in terms of dimensional embedding, however, requires more subtlety. In the next subsection we'll discuss this.

Dimensional Embedding and InheritanceLinks

Next, how can we use dimensional embedding to keep an approximate record of which links do not inherit from each other? Because inheritance is an asymmetric relationship, whereas distance in embedding spaces is a symmetrical relationship, there's no direct and simple way to do so.

However, there is an indirect approach that appears to work, which involves maintaining two different embedding spaces, and combining information about them in an appropriate way. The two embedding spaces correspond to intensional and extensional information.

In this subsection, we'll discuss an approach that should work for InheritanceLink, SubsetLink, ImplicationLink and ExtensionalImplicationLink. But we'll explicitly present it only for the InheritanceLink case.

Although the embedding algorithm described above was intended for symmetric weighted graphs, in fact we can use it for asymmetric links in just about the same way. The use of the embedding graph for inference control differs, but not the basic method of defining the embedding.

In the InheritanceLink case, for the intensional embedding space, we can define pivot Atoms in the same way, and then we can define

```
projint(A)_i
```

to be equal to

```
(InheritanceLink A A_i).tv.s
```

where A_i is the i'th pivot Atom.

Similarly, for the InheritanceLink extensional embedding space, we define

```
projext(A)_i
```

to be equal to

```
(InheritanceLink A_i A).tv.s
```

Define

```
Int(A) = {X : InheritanceLink X A or SubsetLink X A or MemberLink X A)

Ext(A) = {X : InheritanceLink A X or SubsetLink A X}

SimInt(A,B) = ||Int(A) Intersection Int(B) ||/ || Int(A) Union Int(B)

SimExt(A,B) = ||Ext(A) Intersection Ext(B) ||/ || Ext(A) Union Ext(B)
```

The intensional embedding space will then have

```
2^{-cd(proj_int(A), proj_int(B))}
```

correlate with

```
SimInt(A,B)
```

whereas the extensional embedding space will have

```
2^{-cd(proj_ext(A), proj_ext(B))}
```

correlate with

```
SimExt(A,B)
```

Furthermore, if

```
Int(B) within Int(A)
```

then one can say something geometrically about the relationship between projint(A) and projint(B) in the intensional embedding space. Suppose one creates an (n-1)-dimensional hyperplane crossing through all the coordinate axes and also projint(B). This hyperplane divides the embedding space in two halves, one of which contains the origin. The half that does not contain the origin is where projint(A) should ideally lie. Arithmetically, one may compute

```
int_inh(A,B) = Sum_i (||| projint(A)_i ' projint(B)_i |||)
```

as an estimate of the likelihood that

```
InheritanceLink A B
```

Similarly, in the extensional embedding space, one may compute

```
ext_inh(A,B) = Sum_i ( |||projext(B)_i ' projext(A)_i |||)
```

as an estimate of the likelihood that

```
InheritanceLink A B
```

One may then decide whether it's worth evaluating

```
(InheritanceLink A B).tv.s
```

(via InheritanceMining or inference) based on:

```
importance(A) * importance(B) * f(int_inh(A,B), ext_inh(A,B))
```

where for instance we may choose

```
f(x,y) = c*(x+y)
```

where c is a normalizing parameter (that may, for instance, map f(int_inh(A,B), ext_inh(A,B)) into [0,1] for pairs (A,B)

```
distproj(A,B) = d_E( proj(A) , proj(B) )
```

Chapter Thirteen: Probabilistic Evolutionary Learning

Probabilistic Evolutionary Procedure Learning for OpenCog Prime

This page and the ones that it supervenes over

- OpenCogPrime:EvolutionaryPatternMining
- OpenCogPrime:FitnessEstimation
- OpenCogPrime:HierarchicalLearning

- OpenCogPrime:KnowledgeGuidedHierarchicalLearning
- OpenCogPrime:EvolutionBasedInferenceControl
- OpenCogPrime:EvolutionaryLearningWithMemory

deal with various aspects of probabilistic evolutionary learning in OpenCog Prime.

They had their origin in a variety of documents written by Ben Goertzel, Moshe Looks, Cassio Pennachin, and Lucio de Souza Coelho.

Introduction

Evolution is the E in SMEPH, which is one indicator of the central role that evolutionary learning processes play in the cognitive theory underlying OCP. On a SMEPH level, what evolution means is the continual creation of new structures and dynamics from the fragments of old, in a manner driven by system goals and subgoals. I.e. evolution in a SMEPH context is "novelty generation, plus differential reproduction based on fitness, where the entities being reproduced are Edges and Vertices. The idea is that new concepts, procedures and relationships are constantly forming, and these are in large part variations on or combinations of previously existing concepts/procedures/relationships, where the ones chosen for variation/intercombination tend to be the ones that have been involved in successful achievement of system goals.

This kind of evolution can occur in an intelligent system whether or not the low-level *implementation layer* of the system involves any explicitly evolutionary processes. For instance it's clear that the human mind/brain involves evolution in this sense on the emergent level -- we create new ideas and procedures by varying and combining ones that we've found useful in the past, and this occurs on a variety of levels of abstraction in the mind. But the extent to which human neurodynamics display directly evolutionary characteristics is subject to debate. Following Edelman, in designing OpenCog Prime we have come down on the side that believes brain dynamics are intrinsically evolutionary in nature (see Ben Goertzel's book The Evolving Mind), and that neuronal maps (networks of neuronal clusters) are the level on which differential reproduction based on fitness and evolutionary novelty-generation are best conceived to occur.

OCP is intended to display evolutionary dynamics on the derived-hypergraph level, and this is intended to be a consequence of both explicitly-evolutionary and not-explicitly-evolutionary dynamics on the CIM level. Cognitive processes such as PLN inference may lead to emergent evolutionary dynamics (as useful logical relationships are reasoned on and combined, leading to new logical relationships in an evolutionary manner); even though PLN in itself is not explicitly *evolutionary* in character, it becomes emergently evolutionary via its coupling with OCP's attention allocation subsystem, which gives more cognitive attention to Atoms with more importance, and hence creates an evolutionary dynamic with importance as the fitness criterion and the whole constellation of MindAgents as the novelty-generation mechanism. However there is also an "evolutionary learning" subsystem which explicitly embodies evolutionary dynamics for the learning of new patterns and procedures. This evolutionary learning subsystem naturally also contributes to the creating of evolutionary patterns on the emergent, derived-hypergraph level.

The bulk of the pages supervened over by this one deal with the explicitly-evolutionary learning subsystem of OCP, which is focused on the learning of predicates and schemata — i.e. on learning good Combo trees to put inside GroundedProcedureNodes, where *good* is defined in a contextually-appropriate way.

The learning of predicates and schemata is done in OCP via a number of different methods, including for example PLN inference and concept predicatization (to be discussed below). Most of these methods, however, merely extrapolate procedures directly from other procedures or concepts in the Atomspace, in a *local* way — a new procedure is derived from a small number of other procedures or concepts. Intelligence also requires a method for deriving new procedures that are more "fundamentally new." This is where OCP makes recourse to explicitly evolutionary dynamics.

The basic method that OCP uses to learn procedures that appear fundamentally new from the point of view of the Atomspace at a given point in time is "specification-based procedure learning. This involves taking a PredicateNode with a ProcedureNode input type as a specification, and searching for ProcedureNodes that fulfill this specification (in the sense of making the specification PredicateNode as true as possible). In evolutionary computing lingo, the specification predicate is a *fitness function*.

Searching for PredicateNodes that embody patterns in the Atomspace as a whole is a special case of this kind of learning, where the specification PredicateNode embodies a notion of what constitutes an "interesting pattern. The quantification of interestingness is of course an interesting and nontrivial topic in itself ;-)

Finding schemata that are likely to achieve goals important to the system is also a special case of this kind of learning. In this case, the specification predicate is of the form

```
F(S) = PredictiveImplicationLink (ExOutLink S) G
```

This measures the extent to which executing schema S is positively correlated with goal-predicate G being achieved shortly later.

But given a PredicateNode interpretable as a specification, how do we find a ProcedureNode satisfying the specification?We address this problem in OCP using an approach called Probabilistic Evolutionary Learning, specific examples of which are MOSES and Pleasure.

Estimation of Distribution Algorithms for OCP

EDA program learning as applied in OCP involves the probabilistic-inference-guided evolution of Combo trees as described in previous chapters. Currently EDA in OCP is represented by the MOSES algorithm, but extension to Pleasure is envisioned, along with various more ambitious extensions as indicated in the pages supervened by this one.

In OCP-EDA, the *fitness function* is a predicate that takes in a Combo tree and puts out a number in [0,1] telling how *fit* the Combo tree is. Thus we can portray an OCP-EDA fitness function as a PredicateNode with input type ProcedureNode and output type SimpleTruthValue. But OCP-EDA goes further than just applying the EDA idea to Combo trees, incorporating ideas such as

- Using PLN to give OCP-EDA a memory across all the different optimization problems encountered by a OCP system during its history
- Using OCP-EDA as an inference control scheme, to guide the amount of PLN inference applied to various conjectural solutions to a problem during the course of exploratory problem-solving
- Designing modeling schemes that recognize 'fitness-correlated patterns' involving phenotypic as well as genotypic characteristics

These integrative cognitive strategies lie at the heart of deep OCP cognition. They are the key to OCP procedure learning, which is key to OCP intelligence.

Evolution As an Inference Control Scheme

It is possible to use OpenCogPrime:EvolutionaryLearning as, in essence, an OpenCogPrime:InferenceControl scheme. Suppose we are using an evolutionary learning mechanism such as OpenCogPrime:MOSES or OpenCogPrime:Pleasure to evolve populations of predicates or schemata. Recall that there are two ways to evaluate procedures in Novamente: by inference or by direct evaluation. Consider the case where inference is needed in order to provide high-confidence estimates of the evaluation or execution relationships involved. Then, there is the question of how much effort to spend on inference, for each procedure being evaluated as part of the BOA fitness evaluation process. Spending a small amount of effort on inference means that one doesn't discover much beyond what's immediately apparent in the Atomspace. Spending a large amount of effort on inference means that one is trying very hard to use indirect evidence to support conjectures regarding the evaluation or execution Links involved.

When one is evolving a large population of procedures, one can't afford to do too much inference on each candidate procedure being evaluated. Yet, of course, doing more inference may yield more accurate fitness evaluations, hence decreasing the number of fitness evaluations required.

A commonly good heuristic is to gradually increase the amount of inference effort spend on procedure evaluation, during the course of evolution. Specifically, one may make the amount of inference effort roughly proportional to the overall population fitness. This way, initially, evolution is doing a cursory search, not thinking too much about each possibility. But once it has some fairly decent guesses in its population, then it starts thinking hard, applying more inference to each conjecture.

Since the procedures in the population are likely to be interrelated to each other, inferences done on one procedure are likely to produce intermediate knowledge that's useful for doing inference on other procedures. Therefore, what one has in this scheme is evolution as a control mechanism for higher-order inference.

Combined with the use of evolutionary learning to achieve memory across optimization runs, this is a very subtle approach to inference control, quite different from anything in the domain of logic-based AI. Rather than guiding individual inference steps on a detailed basis, this type of control mechanism uses evolutionary logic to guide the general direction of inference, pushing the vast mass of exploratory inferences in the direction of solving the problem at hand, based on a flexible usage of prior knowledge.

Supplying Evolutionary Learning with Long-Term Memory

This page introduces an important enhancement to evolutionary learning, which in a sense violates the basic EDA framework, by forming an adaptive hybridization of EDA optimization with PLN inference (rather than merely using PLN inference within evolutionary learning to aid with modeling). These are brief to state but extremely powerful in their implications.

The first idea is the use of PLN to supply evolutionary learning with a long-term memory. evolutionary learning approaches each problem as an isolated entity, but in reality, a Novamente system will be confronting a long

series of optimization problems, with subtle interrelationships. When trying to optimize the function f, Novamente may make use of its experience in optimizing other functions g.

Inference allows optimizers of g to be analogically transformed into optimizers of f, for instance it allows one to conclude

```
Inheritance f g
    EvaluationLink f x
    EvaluationLink g x
```

However, less obviously, inference also allows patterns in populations of optimizers of g to be analogically transformed into patterns in populations of optimizers of f. For example, if pat is a pattern in good optimizers of f, then we have

```
InheritanceLink f g

ImplicationLink
    EvaluationLink f x
    EvaluationLink pat x

ImplicationLink
    EvaluationLink g x
    EvaluationLink pat x
```

(with appropriate probabilistic truth values), an inference which says that patterns in the population of f-optimizers should also be patterns in the population of g-optimizers).

Note that we can write the previous example more briefly as

```
InheritanceLink f g
    ImplicationLink (EvaluationLink f) (EvaluationLink pat)
    ImplicationLink (EvaluationLink g) (EvaluationLink pat)
```

A similar formula holds for SimilarityLinks.

We may also infer

```
ImplicationLink (EvaluationLink g) (EvaluationLink pat_g)

ImplicationLink (EvaluationLink f) (EvaluationLink pat_f)

ImplicationLink (EvaluationLink (g AND f)) (EvaluationLink (pat_g AND pat_f))
```

and

```
ImplicationLink (EvaluationLink f) (EvaluationLink pat)

ImplicationLink (EvaluationLink ~f) (EvaluationLink ~pat)
```

Through these sorts of inferences, PLN inference can be used to give evolutionary learning a long-term memory, allowing knowledge about population models to be transferred from one optimization problem to another. This complements the more obvious use of inference to transfer knowledge about specific solutions from one optimization problem to another.

Knowledge-Guided Hierarchical Procedure Decomposition

This page described a specific algorithm we have implemented and tested — in a specialized domain somewhat different from AGI, yet not wholly unrelated — which embodies some of the ideas from that section. We will then discuss the conceptual implications of these experiments for hierarchical evolutionary learning in OCP.

What we describe here is an approach for using domain knowledge to break the procedure-learning process down into smaller procedures-learning problems that can be composed hierarchically to form overall procedures. Specifically, the idea is to identify certain *intermediary variables* that are likely to be involved in fit procedures, and then explicitly seek both symbolic regression models that predict the intermediary variables, and procedures that take the intermediary variables as inputs. For instance, in the case study to be described here first, the procedure-learning problem is a classification problem that involves determining whether or not an individual has a given disease based on biological indicators such as gene expression data, and the intermediate variables may reflect the extent to which an individual displays various symptoms of that disease, and other relevant clinical data about the individual. Similar ideas may be used in domains more relevant to OCP AGI, and we will give some hypothetical examples in the simulated-agent-control context.

Formalization of Knowledge-Guided Hierarchical Evolutionary Computing

In order to formulate the relevant technique mathematically we will consider the *function learning* problem in a reasonably general setting. One has a set of input-output pairs

```
{(x_i, y_i), i=1,…,n}
```

where the x_i are drawn from an input space X and the y_i are drawn from an output space Y. One wishes to find a compact and simple function F so that the average of the errors ||y_i — F(x_i)|| is small.

In addition to this standard setup, however, assume that one also has some additional *intermediate values*

```
{z_i; i=1,…,n}
```

In general these values may be taken to lie in some intermediate space Z, which may be different from X or Y. One interesting case is where the z_i are simpler in some sense than the x_i. For instance, the x_i and z_i might both be higher-dimensional vectors, but the dimension of x_i might be much higher than the dimension of z_i.

Given the additional *intermediate values* z_i, one can try to learn a function F that computes the y_i from the x_i via the z_i. That is, one may try to solve the function learning problem of predicting z_i from x_i, and then the problem of predicting y_i from z_i; and then one may chain these predictors together to predict y_i from x_i.

For instance, if each z_i is a numerical vector of length k, then we may say

```
z_i = (z_i1,…,z_ik)
```

and we may seek k separate predictors, each one predicting some z_ij from x_i. These k predictors together give a prediction of z_i from x_i, which may be composed with a predictor of y_i from z_i to solve the original problem.

An important special case of this approach (the one used in our current implementation of the technique described in this section) is where the original problem is formulated as classification rather than function learning. In this case the y_i are either 1 or 0, representing positive or negative category membership. But it's important to note that, in this case, the intermediate problems (predicting z_i or z_ik from x_i) may still be general function learning problems rather than just classification problems.

Obviously, this hierarchical-decomposition *trick* can only work if the intermediate values are judiciously chosen. Choosing the right hierarchical decomposition for any given problem requires significant domain knowledge. In general this problem may be approached via automated reasoning, but in the application to microarray data classification, the hierarchical decomposition emerged immediately and naturally from the dataset.

Finally, we note that in applications to supervised categorization, this approach can actually use any supervised categorization algorithm on the upper level, taking in the intermediary variables as inputs and making categorical decisions based on these. It's most natural conceptually to use evolutionary computing on this layer as well as on the lower, symbolic regression layer, but in doing practical classification problems, we have found that this *pure* approach doesn't always give the best results.

Application to Clinical-Data-Guided Microarray Data Classification

As noted above, we have extensively tested the above approach in the context of microarray data classification. While this application is not specifically germane to AGI, we briefly discuss it here as it provides a very concrete illustration of the power of the methodology for hierarchical breakdown of complex procedure learning problems.

The microarray data classification problem fits into the framework of the previous subsection as follows. Consider for instance the case where we're classifying a set of people according to whether they have some disease or not. Then each index i corresponds to a particular person, and the yi for that person is either 1 or 0 depending on whether they have the disease or not. The xi for that person is the numerical feature vector derived from their microarray gene expression profile (which may be a simple list of the normalized gene expression values, in the most common case.)

And what are the intermediate values? In some microarray data classification problems there are no good candidates available, so the present approach isn't applicable. In some cases however there is a very strong type of intermediary information: clinical data regarding the degree to which the individuals under study display various symptoms associated with the disease in question, as well as additional more basic clinical data regarding age, sex and so forth. In this case, zik may represent the degree to which individual i displays symptom k; or more generally the value of clinical feature k for individual i. Note that there are likely to be only a handful to a few dozen symptoms or other clinical features, but thousands to tens of thousands of entries in the microarray-derived feature vector. So in this case the intermediary layer provides a significant amount of compression.

The application, then, is as follows. Suppose that one tries to apply evolutionary learning to find a rule predicting disease diagnosis based on gene expression data, but the classification accuracy doesn't live up to one's desires. Then one can try to learn functions predicting, for each person, the extent to which they'll display each clinical feature based on their gene expression values. And one can try (using evolutionary computing or other means) to learn a separate function predicting, for each person, whether they'll have the disease or not based on their symptoms and other clinical features. One can then compose the disease-based-on-clinical-features predictor with the clinical-features-based-on-expression predictors to obtain an overall predictor of disease-based-on-clinical-features. One has simplified a difficult learning problem by breaking it down into smaller component learning problems in a hierarchical way, guided by the intuition that symptoms (and, to a lesser extent, other clinical features) are a useful *intermediary* between the gene expression values and the disease.

We have applied this methodology to analyze a recently-obtained dataset regarding a number of individuals with Chronic Fatigue Syndrome, and a number of corresponding controls. This is a particularly appropriate example, since CFS is a complex disorder that is classically defined as a combination of symptoms. Our first approach to analyzing this dataset involved applying genetic programming and support vector machines to learn classification models, using either direct or Extended feature vectors. This yielded reasonable results, but not ideal ones (accuracies in the high 70% range). Then, applying the hierarchical approach described here to the dataset, yielded clearly superior results, including a 100% accurate classifier As it turns out, the best results were obtained here by applying SVM for the upper-layer classification model learning process, and genetic programming based symbolic regression for learning the intermediary clinical features based on the microarray data.

Constructing Hierarchical Programs for Controlling Actions in a Simulated World

Microarray data classification is a nice example of the power of hierarchical decomposition, but it's substantially simpler than most of the procedure learning problems faced by OCP. In this section we'll discuss a more difficult case -- learning programs for controlling actions in a simulated world. As we'll see, in this situation the same basic approach as described above can be used. But things are more complicated because the intermediate variables — the analogues of the *symptoms* in the biomedical application --have variables in them.

Suppose that, instead of trying to learn a model to predict CFS, one is trying to learn a schema that, if executed, will lead to the prediction that some real-world occurrence will come to pass. For instance, the occurrence might be that the red block on the table in the agent's current room is moved into the box in the following room. In this case, the role of the microarray data in the CFS problem is played by the set of near-term perception and action descriptions available to the system (things like *I see the table within 5 units of my body* or *I am moving my arm down*). One wants to find a sequential combination of near-term perception and action descriptions so that, if this combination is true, then this predicts the goal will be achieved. Just as in the CFS example, one wants to find a combination of gene expression values so that, if this combination is present, then this predicts the person will have CFS. The main difference is the temporal aspect of the simulation world problem; but it's worth noting this can be there in the biology domain as well, for instance if one is dealing with microarray time series data rather than static microarray data.

Here the analogue of the symptoms is the library of mid-level perception and action descriptions that the system has available, and that the system judges to be potentially useful in the context of achieving the overall goal.

The judgment of usefulness can be done in terms of probabilities calculated based on the system's history. For instance, since in this case the goal explicitly involves some object beginning in one place and winding up in a different place, simple attention allocation dynamics indicates that schema for *Go to X* should be relevant. This attention allocation works via simple probability calculation. In the past, when achieving goals involving moving things from one place to another (including moving oneself from one place to another), subroutines involving *Go to X* have been commonly used. Therefore, when looking at another goal involving moving, the probability of subroutines of the form *Go to X* being useful is high. Similarly, since the goal involves objects, there is a decent probability that schemata involving picking up objects will be useful.

Once a vocabulary of potentially useful subroutines has been assembled, then the learning process can begin. Note that the learning process is not restricted to the subroutines assembled in advance as potentially promising. It is simply biased in their direction.

The analogue of the CFS strategy, in this context, is to follow a two-stage learning process.

The first stage is to find a schema that, according to reasoning based on experience and logic, will achieve the goal if executed. But express this schema, not in terms of detailed motor actions and perceptions, but in terms of *pseudocode* whose entries are schema such as *Go to the door* and *pick up the box,* i.e. subroutines that look roughly like

```
atTime
  T
  ExecutionLink GoTo door
```

and

```
atTime
  T1
  ExecutionLink PickUp box
```

Here T and T1 are internal VariableNodes referring to time points within the execution of the overall schema; and door and box are internal VariableNodes referring to the door to the other room involved in the goal, and the box involved in the goal.

The trick is that, at the time this higher-level pseudocode schema is learned, the system doesn't really know how it's going to execute GoTo as applied to door. GoTo and PickUp are, here, assumed to be ungrounded SchemaNodes, each one linked to a number of grounded SchemaNodes. If there are no obstacles between the agent and the door at the time the subroutine *GoTo door* is encountered, then the system will be able to use an existing grounded SchemaNode linked to the ungrounded GoTo SchemaNode. On the other hand, if there are complex obstacles in the way, and the system has never had to deal with navigating through a field of complex objects before, then the system will have to learn a new grounding for GoTo in the context of this particular goal-achieving schema in which GoTo appears. Note, however, that this learning problem — the second stage of the original learning problem -- is considerably smaller than the problem of learning a schema to carry out the overall goal G. Just as the problem of learning the overall pseudocode program is smaller than the problem of learning a schema to carry out the overall goal G.

The schema carrying out this particular goal, in informal rather than mathematical form, would look something like:

```
PredictiveImplication
  AND_sequential
    Until see door straight ahead
    Rotate left
    Until distance to door < 3 units
    Move forward
  Achieve goal of finding door
```

On the other hand, for a more complex task such as [G = *move the red block from the table in its current room to the box in the next room*], pseudocode would look more like

```
PredictiveImplication

 ANDsequential

  If possible, locate red block on table  $T
     (using memory of where table $T is located if available, otherwise by
      scanning the room)
   If not possible, locate table $T
   Go to table $T
   If red block was not located on table $T before,  locate red block on table
   Move as close as possible to red block
   Pick up red block
   Locate door to next room (again, using memory if available)
   Go to door
   Go through door slightly
   Look for box
   Go to box
   Put down block in box
  Achieve G
```

Note that this simple pseudocode only works if there's only one table in the room, only one door to another room, and if the box has a big opening that covers the whole top of the box. The knowledge representation is supposed to be flexible enough that generalizing the schema to deal with situations that break these assumptions can be done by localized changes.

This pseudocode schema, fully expanded, will involve about 150-200 links, if implemented with appropriate abstractions but without obsessive attention to compactness. As an example of abstraction: *Go to door* and *Go to box* shouldn't be implemented as totally separate subprograms; rather there should be a subprogram *Go to X* and both of the special cases should be implemented in terms of this.

The higher-level learning process — the one that learns the pseudocode program — can be carried out by BOA learning on Combo trees. However, in this case (unlike the CFS case) the fitness function for this evolutionary learning process must be inferential, since at this stage the system doesn't know exactly how the subcomponents will be carried out. The lower-level learning processes may be carried out by evolutionary learning as well, and the fitness evaluation will be inferential sometimes and direct-evaluational other times, depending on the context.

In some cases, of course, one may have multiple levels of pseudocode, with several levels of abstraction before one gets down to the level of particular perceptions and actions.

Although the discussion has been somewhat abstract, we hope the basic concept of the approach is clear. One learns hierarchical schemata by design, using inferential fitness evaluation to learn the higher levels before the lower levels have been filled in. This is a fairly simple *cognitive schema,* but we believe it's an essential one.

Hierarchical EDA Program Learning

This page discusses hierarchical program structure, and its reflection in probabilistic modeling. This is a surprisingly subtle and critical topic, which may be approached from several different complementary angles.

In human-created software projects, one common approach for dealing with the existence of complex interdependencies between parts of a program is to give the program a hierarchical structure. The program is then a hierarchical arrangement of programs within programs within programs, each one of which has relatively simple dependencies between its parts (however its parts may themselves be hierarchical composites). This notion of hierarchy is essential to such programming methodologies as modular programming and object-oriented design.

Pelikan and Goldberg discuss the hierarchal nature of human problem-solving, in the context of the hBOA (hierarchical BOA&) version of BOA. However, the hBOA algorithm does not incorporate hierarchical program structure nearly as deeply and thoroughly as the hierarchical BOA approach proposed here. In hBOA the hierarchy is implicit in the models of the evolving population, but the population instances themselves are not necessarily explicitly hierarchical in structure. In hierarchical NM-PEL as we describe it here, the population consists of hierarchically structured Combo trees, and the hierarchy of the probabilistic models corresponds directly to this hierarchical program structure.

The ideas presented here have some commonalities to John Koza's ADFs and related tricks for putting reusable subroutines in GP trees, but there are also some very substantial differences, which we believe will make the current approach far more effective (though also considerably more computationally expensive).

We believe that this sort of hierarchically-savvy modeling is what will be needed to get probabilistic evolutionary learning to scale to large and complex programs, just as hierarchy-based methodologies like modular and object-oriented programming are needed to get human software engineering to scale to large and complex programs.

Hierarchical Modeling of Composite Procedures in the Atomspace

The possibility of hierarchically structured programs is (intentionally) present in the Novamente design, even without any special effort to build hierarchy into the NM-PEL framework. Combo trees may contain NMNodes that point to PredicateNodes, which may in turn contain Combo trees, etc. However, our current BOA framework for learning Combo trees does not take advantage of this hierarchality. What is needed, in order to do so, is for the models used for instance generation to include events of the form

Combo tree Node at position x has type PredicateNode; and the PredicateNode at position x contains a Combo tree that possesses property P

where x is a position in a Combo tree and P is a property that may or may not be true of any given Combo tree. Using events like this, a relatively small BOA model explicitly incorporating only *short-range dependencies*; may implicitly encapsulate long-range dependencies via the properties P.

But where do these properties P come from? These properties should be patterns learned as part of the probabilistic modeling of the Combo tree inside the PredicateNode at position x. For example, if one is using a decision tree modeling framework, then the properties might be of the form *decision tree D evaluates to True*. Note that not all of these properties have to be statistically correlated with the fitness of the PredicateNode at position x (although some of them surely will be).

Thus we have a multi-level probabilistic modeling strategy. The top-level Combo tree has a probabilistic model whose events may refer to patterns that are parts of the probabilistic models of Combo trees that occur within it and so on down.

In instance generation, when a newly generated Combo tree is given a PredicateNode at position x, two possibilities exist:

- There is already a model for PredicateNodes at position x in Combo trees in the given population, in which case a population of PredicateNodes potentially living at that position are drawn from the known model, and evaluated
- There is no such model (because it has never been tried to create a PredicateNode at position x in this population before), in which case a new population of Combo trees is created corresponding to the position, and evaluated

Note that the fitness of a Combo tree that is not at the top level of the overall process, is assessed indirectly in terms of the fitness of the higher-level Combo tree in which it is embedded.

Suppose each Combo tree in the hierarchy has on average R adaptable sub-programs (represented as NMNodes pointing to PredicateNodes containing Combo trees to be learned). Suppose the hierarchy is K levels deep. Then we will have about R*K PEL populations in the tree. This suggests that hierarchies shouldn't get too big, and indeed, they shouldn't need to, for same essential reason that human-created software programs, if well-designed, tend not to require extremely deep and complex hierarchical structures.

One may also introduce a notion of *reusable components* across various program learning runs, or across several portions of the same hierarchical program. Here one is learning patterns of the form:

```
If property P1(C,x) applies to a Combo tree C and a node x within it,
then it is often good for node x to refer to a PredicateNode
containing a Combo tree with property P2
```

These patterns may be assigned probabilities and may be used in instance generation. They are general or specialized *programming guidelines,* which may be learned over time.

Identifying Hierarchical Structure In Combo trees via MetaNodes and Dimensional Embedding

One may also apply the concepts of the previous section to model a population of CTs that doesn't explicitly have an hierarchical structure, via introducing the hierarchical structure during the evolutionary process, through the introduction of special extra Combo tree nodes called MetaNodes. This may be done in a couple

different ways, here we will introduce a simple way of doing this based on dimensional embedding, and then in the next section we will allude to a more sophisticated approach that uses inference instead.

The basic idea is to couple decision tree modeling with dimensional embedding of subtrees, a trick that enables small decision tree models to cover large regions of a CT in an approximate way, and which leads naturally to a form of probabilistically-guided crossover.

The approach as described here works most simply for CT's that have many subtrees that can be viewed as mapping numerical inputs into numerical outputs. There are clear generalizations to other sorts of CT's, but it seems advisable to test the approach on this relatively simple case first.

The first part of the idea is to represent subtrees of a CT as numerical vectors in a relatively low-dimensional space (say N=50 dimensions). This can be done using our existing dimensional embedding algorithm, which maps any metric space of entities into a dimensional space. All that's required is that we define a way of measuring distance between subtrees. If we look at subtrees with numerical inputs and outputs, this is easy. Such a subtree can be viewed as a function mapping Rn into Rm; and there are many standard ways to calculate the distance between two functions of this sort (for instance one can make a Monte Carlo estimate of the Lp metric which is defined as

```
[Sum{x} (f(x) - g(x))^p] ^ (1/p)
```

Of course, the same idea works for subtrees with non-numerical inputs and outputs, the tuning and implementation are just a little trickier.

Next, one can augment a CT with meta-nodes that correspond to subtrees. Each meta-node is of a special CT node type MetaNode, and comes tagged with an N-dimensional vector. Exactly which subtrees to replace with MetaNodes is an interesting question that must be solved via some heuristics.

Then, in the course of executing the BOA algorithm, one does decision tree modeling as usual, but making use of MetaNodes as well as ordinary CT nodes. The modeling of MetaNodes is quite similar to the modeling of NMNodes representing ConceptNodes and PredicateNodes using embedding vectors. In this way, one can use standard, small decision tree models to model fairly large portions of CT's (because portions of CT's are approximately represented by MetaNodes).

But how does one do instance generation, in this scheme? What happens when one tries to do instance generation using a model that predicts a MetaNode existing in a certain location in a CT? Then, the instance generation process has got to find some CT subtree to put in the place where the MetaNode is predicted. It needs to find a subtree whose corresponding embedding vector is close to the embedding vector stored in the MetaNode. But how can it find such a subtree?

There seem to be two ways.

1. A reasonable solution is to look at the database of subtrees that have been seen before in the evolving population, and choose one from this database, with the probability of choosing subtree X being proportional to the distance between X's embedding vector and the embedding vector stored in the MetaNode.
2. One can simply choose good subtrees, where the goodness of a subtree is judged by the average fitness of the instances containing the target subtree.

One can use a combination of both of these processes during instance generation.

But of course, what this means is that we're in a sense doing a form of crossover, because we're generating new instances that combine subtrees from previous instances. But we're combining subtrees in a judicious way guided by probabilistic modeling, rather than in a random way as in GP-style crossover.

Inferential MetaNodes

MetaNodes are an interesting and potentially powerful technique, but we don't believe that they, or any other algorithmic trick, is going to be the solution to the problem of learning hierarchical procedures. We believe that this is a cognitive science problem that probably isn't amenable to a purely computer science oriented solution. In other words, we suspect that the correct way to break a Combo tree down into hierarchical components depends on context, algorithms are of course required, but they're algorithms for relating a CT to its context rather than pure CT-manipulation algorithms. Dimensional embedding is arguably a tool for capturing contextual relationships, but it's a very crude one.

Generally speaking, what we need to be learning are patterns of the form "A subtree meeting requirements X is often fit when linked to a subtree meeting requirements Y, when solving a problem of type Z". Here the context requirements Y will not pertain to absolute tree position but rather to abstract properties of a subtree.

The MetaNode approach as outlined above is a kind of halfway measure toward this goal, good because of its relative computational efficiency, but ultimately too limited in its power to deal with really hard hierarchical learning problems. The reason the MetaNode approach is crude is simply because it involves describing subtrees via points in an embedding space. We believe that the *correct* (but computationally expensive) approach is indeed to use MetaNodes --- but with each MetaNode tagged, not with coordinates in an embedding space, but with a set of logical relationships describing the subtree that the MetaNode stands for. A candidate subtree's similarity to the MetaNode may then be determined by inference rather than by the simple computation of a distance between points in the embedding space. (And, note that we may have a hierarchy of MetaNodes, with small subtrees corresponding to MetaNodes, larger subtrees comprising networks of small subtrees also corresponding to MetaNodes, etc.)

The question then becomes which logical relationships one try to look for, when characterizing a MetaNode. This may be partially domain-specific, in the sense that different properties will be interesting when studying motor-control procedures than when studying cognitive procedures.

To intuitively understand the nature of this idea, let's consider some abstract but commonsensical examples. Firstly, suppose one is learning procedures for serving a ball in tennis. Suppose all the successful procedures work by first throwing the ball up really high, then doing other stuff. The internal details of the different procedures for throwing the ball up really high may be wildly different. What we need is to learn the pattern

```
Implication

      Inheritance X "throwing the ball up really high"

      "X then Y" is fit
```

Here X and Y are MetaNodes. But the question is how do we learn to break trees down into MetaNodes according to the formula " tree='X then Y' where X inherits from 'throwing the ball up really high.'"?

Similarly, suppose one is learning procedures to do first-order inference. What we need is to learn a pattern such as

```
Implication

      AND

            F involves grabbing pairs from the AtomTable

            G involves applying an inference rule to those each pair

            H involves putting the results back in the AtomTable

      "F ( G (H)))" is fit
```

Here we need MetaNodes for F, G and H, but we need to characterize e.g. the MetaNode F by a relationship such as "involves grabbing pairs from the AtomTable."

Until we can characterize MetaNodes using abstract descriptors like this, we're just doing simplistic statistical learning rather than "general intelligence style" procedure learning. But to do this kind of abstraction intelligently seems to require some background knowledge about the domain.

In the "throwing the ball up really high" case the assigment of a descriptive relationship to a subtree involves looking, not at the internals of the subtree itself, but at the state of the world after the subtree has been executed.

In the "grabbing pairs from the AtomTable" case it's a bit simpler but still requires some kind of abstract model of what the subtree is doing, i.e. a model involving a logic expression such as "The output of F is a set S so that if P belongs to S then P is a set of two Atoms A1 and A2, and both A1 and A2 were produced via the getAtom operator."

How can this kind of abstraction be learned? It seems unlikely that abstractions like this will be found via evolutionary search over the space of all possible predicates describing program subtrees. Rather, they need to be found via probabilistic reasoning based on the terms combined in subtrees, put together with background knowledge about the domain in which the fitness function exists. In short, integrative cognition is required to learn hierarchically structured programs in a truly effective way, because the appropriate hierarchical breakdowns are contextual in nature, and to search for appropriate hierarchical breakdowns without using inference to take context into account, involves intractably large search spaces.

Fitness Estimation via Integrative Intelligence

If instance generation is very cheap and fitness evaluation is very expensive (as is the case in many applications of evolutionary learning in the Novamente context), one can accelerate evolutionary learning via a 'fitness

estimation' approach. Given a fitness function embodied in a predicate P, the goal is to learn a predicate Q so that

1. Q is much cheaper than P to evaluate, and
2. There is a high-strength relationship

```
Similarity Q P

or else

ContextLink C (Similarity Q P)

where C is a relevant context
```

Given such a predicate Q, one could proceed to optimize P by ignoring evolutionary learning altogether and just repeatedly following the algorithm:

- Randomly generate N candidate solutions
- Evaluate each of the N candidate solutions according to Q
- Take the k<<N solutions that satisfy Q best, and evaluate them according to P

improved based on the new evaluations of P that are done. Of course, this would not be as good as incorporating fitness estimation into an overall evolutionary learning framework, however.

Heavy utilization of fitness estimation may be appropriate, for example, if the entities being evolved are schemata intended to control an agent's actions in a real or simulated environment. In this case the specification predicate P, in order to evaluate P(S), has to actually use the schema S to control the agent in the environment. So one may search for Q that do not involve any simulated environment, but are constrained to be relatively small predicates involving only cheap-to-evaluate terms (e.g. one may allow standard combinators, numbers, strings, ConceptNodes, and predicates built up recursively from these). Then Q will be an abstract predictor of concrete environment success.

We have left open the all-important question of how to find the 'specification approximating predicate' Q.

One approach is to use evolutionary learning. In this case, one has a population of predicates, which are candidates for Q. The fitness of each candidate Q is judged by how well it approximates P over the set of candidate solutions for P that have already been evaluated. If one uses evolutionary learning to evolve Q's, then one is learning a probabilistic model of the set of Q's, which tries to predict what sort of Q's will better solve the optimization problem of approximating P's behavior. Of course, using evolutionary learning for this purpose potentially initiates an infinite regress, but the regress can be stopped by, at some level, finding Q's using a non-evolutionary learning based technique such as genetic programming, or a simple evolutionary learning based technique like standard BOA programming.

Another approach to finding Q is to use inference based on background knowledge. Of course, this is complementary rather than contradictory to using evolutionary learning for finding Q. There may be information in the knowledge base that can be used to "analogize" regarding which Q's may match P. Indeed, this will generally be the case in the example given above, where P involves controlling actions in a simulated environment but Q does not.

An important point is that, if one uses a certain Q1 within fitness estimation, the evidence one gains by trying Q1 on numerous fitness cases may be utilized in future inferences regarding other Q2 that may serve the role of Q. So, once inference gets into the picture, the quality of fitness estimators may progressively improve via ongoing analogical inference based on the internal structures of the previously attempted fitness estimators.

Pattern Mining via Evolutionary Learning

There are two major applications of evolutionary learning in OCP:

- procedure learning (learning procedures to control behaviors in AGISim, or control schemata for processes like inference or attention allocation)
- pattern mining — finding predicates recognizing patterns that are observable in the Atomspace.

Within the scope of pattern mining, there are two approaches

- supervised learning: given a predicate, finding a pattern among the entities that satisfy that predicate
- unsupervised learning: undirected search for "interesting patterns"

The supervised learning case is easier and we have done a number of experiments using MOSES for supervised learning, on biological (microarray gene expression and SNP) and textual data. In the OCP application, the "positive examples" are the elements of the SatisfyingSet of the predicate P, and the "negative examples" are everything else. This can be a relatively straightforward problem if there are enough positive examples and they actually share common aspects ... the trickiness comes of course when the common aspects are, in each example, complexly intertwined with other aspects.

The unsupervised learning case is trickier. The main problem issue here regards the definition of an appropriate fitness function. We are searching for "interesting patterns." So the question is, what constitutes an interesting pattern?

Evolving Interesting Patterns

Clearly, "interestingness" is a multidimensional concept. One approach to defining interestingness is empirical, based on observation of which predicates have and have not proved interesting to the system in the past (based on their long-term importance values, i.e. LTI).

In this approach, one has a supervised categorization problem: learn a rule predicting whether a predicate will fall into the *interesting* category or the *uninteresting* category. Once one has learned this rule, which is expressed this rule as a predicate itself, one can then use this rule as the fitness function for evolutionary learning evolution.

There is also a simpler approach, which defines an *objective* notion of interestingness. This objective notion is a weighted sum of two factors:

- Compactness
- Surprisingness of truth value

Compactness is easy to understand: all else equal, a predicate embodied in a small Combo tree is better than a predicate embodied in a big one. There is some work hidden here in Combo tree reduction; ideally, one would like to find the smallest representation of a given Combo tree, but this is a computationally formidable problem, so one necessarily approaches it via heuristic algorithms.

Surprisingness of truth value is a slightly subtler concept. Given a predicate, one can envision two extreme ways of evaluating its truth value (represented by two different types of ProcedureEvaluator). One can use an IndependenceAssumingProcedureEvaluator, which deals with all AND and OR operators by assuming probabilistic independence. Or, one can use an ordinary EffortBasedProcedureEvaluator, which uses dependency information wherever feasible to evaluate the truth values of AND and OR operators. These two approaches will normally give different truth values — but, how different? The more different, the more *surprising* is the truth value of the predicate, and the more *interesting* may the predicate be.

In order to explore the power of this kind of approach, we have tested pattern mining using BOA on Boolean predicates as a data mining algorithm on a number of different datasets, including some interesting and successful work in the analysis of gene expression data, and some more experimental work analyzing sociological data from the National Longitudinal Survey of Youth (NLSY).

A very simple illustrative result from the analysis of the NLSY data is the pattern:

```
OR
   (NOT(MothersAge(X)) AND NOT(FirstSexAge(X)))
   (Wealth(X) AND PIAT(X))
```

meaning that:

- being the child of a young mother correlates with having sex at a younger age;
- being in a wealthier family correlates with better Math (PIAT) scores;
- the two sets previously described tend to be disjoint.

Of course, many data patterns are several times more complex than the simple illustrative pattern shown above. However, one of the strengths of the evolutionary learning approach to pattern mining is its ability to find simple patterns when they do exist, yet without (like some other mining methods) imposing any specific restrictions on the pattern format.

Chapter Fourteen: Speculative Concept Formation

Speculative Concept Formation

- OpenCogPrime:EvolutionaryConceptFormation
- OpenCogPrime:Clustering

OpenCog Prime nodes and SMEPH vertices represent concepts, percepts and actions. Clearly a mind cannot rely on a fixed set of these things — so, creation of new nodes/vertices is critical.

The philosophy underlying OCP concept formation is that new things should be created from pieces of good old things (evolution), and that probabilistic extrapolation from experience should be used to guide the creation of

new things (inference). It's clear that these principles are necessary for the creation of new mental forms but it's not obvious that they're sufficient: this is a nontrivial hypothesis, which may also be considered a family of hypotheses since there are many different ways to do extrapolation and intercombination.

In OpenCog Prime we have introduced a variety of heuristics for creating new Atoms - especially ConceptNodes — which may then be reasoned on and subjected to implicit (via attention allocation) and explicit (via the application of evolutionary learning to predicates obtained from concepts via "concept predicatization") evolution. Probably the most valuable of these heuristics are the node logical operators described in Probabilistic Logic Networks, which allow the creation of new concepts via AND, OR, XOR and so forth. However, logical heuristics alone are not sufficient. In this chapter we will review some of the nonlogical heuristics that are used for speculative concept formation. These operations play an important role in creativity — to use cognitive-psychology language, they are one of the ways that OpenCog Prime implements the process of blending, which Falconnier and Turner (2003) have argued is key to human creativity on many different levels.

An evolutionary perspective may be useful here, on a technical level as well as philosophically. As discussed in Ben Goertzel's book The Hidden Pattern (following up on earlier, conceptual ideas from his 1993 book The Evolving Mind), one way to think about OpenCog Prime is as a huge evolving ecology. The Atomspace is a biosphere of sorts, and the mapping from Atom types into species has some validity to it (though not complete accuracy: Atom types do not compete with each other; but they do reproduce with each other, and according to most of the reproduction methods in use, Atoms of differing type cannot cross-reproduce). Fitness is defined by importance. Reproduction is defined by various operators that produce new Atoms from old, including the ones discussed in this chapter, as well as other operators such as inference and explicit evolutionary operators.

New node creation may be triggered by a variety of circumstances. Consider the case of ConceptNodes. If two ConceptNodes are created for different purposes, but later the system finds that most of their meanings overlap, then it may be more efficient to merge the two into one. On the other hand, a node may become overloaded with different usages, and it is more useful to split it into multiple nodes, each with a more consistent content. Finally, there may be patterns across large numbers of nodes that merit encapsulation in individual nodes. For instance, if there are 1000 fairly similar ConceptNodes, it may be better not to merge them all together, but rather to create a single node to which they all link, reifying the category that they collectively embody.

In the pages supervened by this one, we will begin by describing operations that create new ConceptNodes from existing ones on a local basis: by mutating individual ConceptNodes or combining pairs of ConceptNodes. Some of these operations are inspired by evolutionary operators used in the GA, others are derived based on inferential semantics. Then we will turn to the use of traditional clustering algorithms inside OpenCog Prime to refine the system's knowledge about existing concepts, and create new concepts.

As of July 2008, the ideas in this chapter have been partially implemented but nowhere near fully explored: a prototype implementation of evolutionary concept formation exists, and clustering on certain types of nodes is being heavily used in bioinformatics applications of the system.

- OpenCogPrime:EvolutionaryConceptFormation
- OpenCogPrime:Clustering

Evolutionary Concept Formation

A simple and useful way to combine ConceptNodes is to use GA-inspired evolutionary operators: crossover and mutation. In mutation, one replaces some number of a Node's links with other links in the system. In crossover, one takes two nodes and creates a new node containing some links from one and some links from another.

More concretely, to cross over two ConceptNodes X and Y, one may proceed as follows:

- Create a series of empty nodes Z1, Z2,.., Zk
- Choose a number w in [0,1]
- Form a "link pool" consisting of all X's links and all Y's links, and then divide this pool into clusters (clustering algorithms will be described below).
- For each cluster with significant cohesion, allocate the links in that cluster to one of the new nodes Zi

On the other hand, to mutate a ConceptNode, a number of different mutation processes are reasonable. For instance, one can

- Cluster the links of a Node, and remove one or more of the clusters, creating a node with less links
- Cluster the links, remove one or more clusters, and then add new links that are similar to the links in the remaining clusters

The EvolutionaryConceptFormation CIM-Dynamic selects pairs of nodes from the system, where the probability of selecting a pair is determined by

- the average importance of the pair
- the degree of similarity of the pair
- the degree of association of the pair

(Of course, other heuristics are possible too). It then crosses over the pair, and mutates the result.

Note that, unlike in some GA implementations, the parent node(s) are retained within the system; they are not replaced by the children. Regardless of how many offspring they generate by what methods, and regardless of their age, all Nodes compete and cooperate freely forever according to the fitness criterion defined by the importance updating function. The entire Atomspace may be interpreted as a large evolutionary, ecological system, and the action of Novamente dynamics, as a whole, is to create fit nodes.

A more advanced variant of the EvolutionaryConceptFormation CIM-Dynamic would adapt its mutation rate in a context-dependent way. But our intuition is that it is best to leave this kind of refinement for learned cognitive schemata, rather than to hard-wire it into a CIM-Dynamic. To encourage the formation of such schemata, we introduce elementary schema functions that embody the basic node-level evolutionary operators:

```
ConceptNode ConceptCrossover(ConceptNode A, ConceptNode B)

ConceptNode mutate(ConceptNode A, mutationAmount m)
```

There will also be a role for more abstract schemata that utilize these. An example cognitive schema of this sort would be one that said: "When all my schema in a certain context seem unable to achieve their goals, then

maybe I need new concepts in this context, so I should increase the rate of concept mutation and crossover, hoping to trigger some useful concept formation."

As noted above, this component of Novamente views the whole Atomspace as a kind of genetic algorithm — but the fitness function is "ecological" rather than fixed, and of course the crossover and mutation operators are highly specialized. Most of the concepts produced through evolutionary operations are going to be useless nonsense, but will be recognized by the importance updating process and subsequently forgotten from the system. The useful ones will link into other concepts and become ongoing aspects of the system's mind. The importance updating process amounts to fitness evaluation, and it depends implicitly on the sum total of the cognitive processes going on in Novamente.

To ensure that importance updating properly functions as fitness evaluation, it is critical that evolutionarily-created concepts (and other speculatively created Atoms) always comprise a small percentage of the total concepts in the system. This guarantees that importance will serve as a meaningful "fitness function" for newly created ConceptNodes. The reason for this is that the importance measures how useful the newly created node is, in the context of the previously existing Atoms. If there are too many speculative, possibly useless new ConceptNodes in the system at once, the importance becomes an extremely noisy fitness measure, as it's largely measuring the degree to which instances of new nonsense fit in with other instances of new nonsense. One may find interesting self-organizing phenomena in this way, but in an AGI context we are not interested in undirected spontaneous pattern-formation, but rather in harnessing self-organizing phenomena toward system goals. And the latter is achieved by having a modest but not overwhelming amount of speculative new nodes entering into the system.

Finally, as discussed earlier, evolutionary operations on maps may occur naturally and automatically as a consequence of other cognitive operations. Maps are continually mutated due to fluctuations in system dynamics; and maps may combine with other maps with which they overlap, as a consequence of the nonlinear properties of activation spreading and importance updating. Map-level evolutionary operations are not closely tied to their Atom-level counterparts (a difference from e.g. the close correspondence between map-level logical operations and underlying Atom-level logical operations).

Clustering

Finally, a different method for creating new ConceptNodes in OCP is using clustering algorithms. There are many different clustering algorithms in the statistics and data mining literature, and no doubt many of them could have value inside OCP. We have experimented with several different clustering algorithms in the OCP context, and have selected one, which we call Omniclust, based on its generally robust performance on high-volume, *noisy* data.

In the discussion on OpenCogPrime:EvolutionaryConceptCreation, we mentioned the use of a clustering algorithm to cluster links. The same algorithm we describe here for clustering ConceptNodes directly and creating new ConceptNodes representing these clusters, can also be used for clustering links in the context of node mutation and crossover.

The application of Omniclust or any other clustering algorithm for ConceptNode creation in OCP is simple. The clustering algorithm is run periodically, and the most significant clusters that it finds are embodied as ConceptNodes, with InheritanceLinks to their members. If these significant clusters have subclusters also

identified by Omniclust, then these subclusters are also made into ConceptNodes, etc., with InheritanceLinks between clusters and subclusters.

Clustering technology is famously unreliable, but this unreliability may be mitigated somewhat by using clusters as initial guesses at concepts, and using other methods to refine the clusters into more useful concepts. For instance, a cluster may be interpreted as a disjunctive predicate, and a search may be made to determine sub-disjunctions about which interesting OpenCogPrime:PLN conclusions may be drawn.

Chapter Fifteen: Integrative Procedure and Predicate Learning

Integrative Procedure Learning

- OpenCogPrime:PredicateSchematization
- OpenCogPrime:ConceptDrivenProcedureLearning
- OpenCogPrime:ProcedureEvolution
- OpenCogPrime:PredicateMining
- OpenCogPrime:SchemaMaps
- OpenCogPrime:OccamsRazor

This page groups a set of pages that discuss two closely related aspects of OCP: schema learning and predicate learning, grouped under the general category of "procedure learning." The material is integrative — we explain how various ideas presented in previous chapters may be used to provide a powerful integrative-intelligence approach to the learning of complex declarative and procedural knowledge. Much of this material is "advanced AGI" which does not yet exist in the current version of OpenCog, but will be critical for the realization of OpenCog's ultimate ambitions.

Schema learning — the learning of SchemaNodes and schema maps — is OCP lingo for learning how to do things. Learning how to act, how to perceive, and how to think — beyond what's explicitly encoded in the system's CIM-Dynamics. Ultimately, when OCP becomes more profoundly self-modifying, schema learning will be the main process driving its evolution.

Predicate learning, on the other hand, is the most abstract and general manifestation of pattern recognition in the OCP system. PredicateNodes, along with predicate maps, are OCP's way of representing general patterns (*general* within the constraints imposed by the system parameters, which in turn are governed by hardware constraints). Predicate evolution, predicate mining and higher-order inference — specialized and powerful forms of predicate learning — are the system's most powerful ways of creating general patterns in the world and in the mind. Simpler forms of predicate learning are grist for the mill of these processes.

All aspects of OCP are difficult, but some are more difficult than others — and procedure learning definitely falls into the latter category. Specifically, of all the AI tasks with which we have experimented, schema learning has proved the trickiest. As the chapter unfolds, we will try to convey the nature of this difficulty, along with the solutions we suggest and why we believe they will be successful.

It may be useful to draw an analogy with another (closely related) very hard problem in OCP, probabilistic logical unification (which in the OCP framework basically comes down to finding the SatisfyingSets of given predicates). Hard logical unification problems can often be avoided by breaking down large predicates into

small ones in strategic ways, guided by non-inferential mind processes, and then doing unification only on the smaller predicates. Our limited experimental experience indicates that the same "hierarchical breakdown" strategy also works for schema and predicate learning, to an extent. But still, as with unification, even when one does break down a large schema or predicate learning problem into a set of smaller problems, one is still in most cases left with a set of fairly hard problems.

More concretely, OCP procedure learning may be generally decomposed into three aspects:

1. Converting back and forth between maps and compound ProcedureNodes (*encapsulation and expansion*)
2. Learning the Combo Trees to be embedded in grounded ProcedureNodes
3. Learning procedure maps (networks of grounded ProcedureNodes acting in a coordinated way to carry out procedures)

These difficult problems are dealt with using a combination of techniques:

- Probabilistic inference (PLN)
- Evolutionary programming (PEL)
- Reinforcement learning (i.e. inference on HebbianLinks)
- Procedure map encapsulation and expansion
- Pattern-formation-driven predicate evolution

We are relying on a combination of techniques to do what none of the techniques can accomplish on their own. The combination is far from arbitrary, however. As we will see, each of the techniques involved plays a unique and important role.

Each of the three aspects of OCP procedure learning mentioned above may be dealt with somewhat separately, though relying on largely overlapping methods.

First, the conversion aspects will be dealt with later on in the more general context of map encapsulation and expansion. However a few comments on procedure encapsulation and expansion may be useful here.

Converting from grounded ProcedureNodes into maps is a relatively simple learning problem. The challenge is choosing among the many possible ways to split a Combo tree into subtrees. Once this is decided, it's easy enough to create separate ProcedureNodes corresponding to the separate subtrees. Determining when to expand compound ProcedureNodes into maps can also be subtle; the process is guided by Goal Nodes and will be discussed later on.

Converting from maps into ProcedureNodes is significantly trickier. First, it involves carrying out datamining over the network of ProcedureNodes, identifying subnetworks that are coherent schema or predicate maps. Then it involves translating the control structure of the map into explicit logical form, so that the encapsulated version will follow the same order of execution as the map version. This is an important case of the general process of map encapsulation, to be discussed in the final chapter.

Next, the learning of grounded ProcedureNodes is carried out by a synergistic combination of two mechanisms: evolutionary programming, and logical inference. These two approaches have quite different characteristics. Evolutionary programming excels at confronting a problem that the system has no clue about, and arriving at a reasonably good solution in the form of a schema or predicate. Inference excels at deploying the system's

existing knowledge to form useful schemata or predicates. The choice of the appropriate mechanism for a given problem instance depends largely on how much relevant knowledge is available.

A relatively simple case of compound ProcedureNode learning is where one is given a ConceptNode and wants to find a ProcedureNode matching it. For instance, given a ConceptNode C, one may wish to find the simplest possible predicate whose corresponding PredicateNode P satisfies

```
SatisfyingSet(P) = C
```

On the other hand, given a ConceptNode C involved in inferred ExecutionLinks of the form

```
ExecutionLink C Ai Bi
i=1,...,n
```

one may wish to find a SchemaNode so that the corresponding SchemaNode will fulfill this same set of ExecutionLinks. These kinds of compound ProcedureNode learning may be cast as optimization problems, and carried out by evolutionary programming. Once learned via evolutionary programming or other techniques, they may be refined via inference.

The other case of compound ProcedureNode learning is goal-driven learning. Here one seeks a SchemaNode whose execution will cause a given goal (represented by a Goal Node) to be satisfied. The details of Goal Nodes have already been reviewed; but all we need to know here is simply that a Goal Node presents an objective function, a function to be maximized; and that it poses the problem of finding schemata whose enaction will cause this function to be maximized.

The learning of procedure maps, on the other hand, is carried out by reinforcement learning. This is a matter of the system learning HebbianLinks between ProcedureNodes, as described in the section Learning Schema Maps.

Note that compound ProcedureNodes learned via evolution and refined via inference may be translated into procedure maps; or, procedure maps learned via reinforcement learning may be translated into compound ProcedureNodes.

Comparison with Other Approaches To Procedure Learning

It is perhaps worth briefly drawing some contrasts with known alternative approaches. Instead of a complete literature review, we merely provide a selective discussion of some alternatives that have seemed interesting to the authors for one reason or another.

One alternative, well known to the original author of this page and several colleagues (including members of the Novamente team) if not to many others, is the Webmind 2000 schema module. This design will not be described here in detail, but some brief indications may be useful. In this approach, complex, learned schema were represented as distributed networks of elementary SchemaInstanceNodes, but these networks were not defined purely by function application - they involved explicit passing of variable values through VariableNodes. Special logic-gate-bearing objects were created to deal with the distinction between arguments of a SchemaInstanceNode, and *predecessor* tokens giving a SchemaInstanceNode permission to act. The basic problem, apart from implementational inefficiency, was a severe brittleness. Small modifications to an effective

schema map would too frequently yield a completely dysfunctional schema. The main problem here was the overcomplexity of the design. Everyone who worked with this system, theoretically or pragmatically, clearly realized that it was infeasibly complex in terms of the human effort required to get all the parts to work together. There seems little doubt that OCP's Combo tree based approach is preferable. The difference is that in OCP, procedural knowledge is stored and manipulated separately from declarative knowledge; whereas in Webmind 2000, procedural knowledge was essentially treated as a special case of declarative and associational, Hebbian-type knowledge, which is mathematically and conceptually correct, but unacceptably inefficient. It turns out that imitating the human brain's decision to separate declarative and procedural knowledge, as an abstract design choice, is a solid idea in terms of achieving design tractability and computational feasibility.

Next, let us consider the procedure-learning methods preferred by the traditional AI community. (Note: this review is extremely brief and inadequate, and should probably be spun out into a separate page at some point. On the other hand, the reviews of AI planning on wikipedia and elsewhere online are not written from an AGI perspective. Someone should write a constructively-critical review of planning software and algorithms from an AGI perspective.)

There are numerous planning methods out there, based on logical inference, graph analysis, simplified probabilistic inference and other techniques — but overall, the performance of these systems is not too impressive. These programs only work effectively given very particular constraints as to the problems they are posed. And almost none of them deal adequately with uncertainty. In OCP terms, these approaches are basically specialized inference control processes. Among our favorites among the existing AI planning approaches is Probabilistic GraphPlan (Avrim and Langford, 1999), but even this is too oversimplified to deal with most real-world planning problems.

There are neural-net-based procedure learning methods, most notably various variations on recurrent backpropagation. But recurrent backprop is an inefficient algorithm, not really scalable to large learning problems (though attempts have been made to work around this issue by recourse to massively parallel implementations; see e.g. Suresh et al, 2005). [fuckwad4] The problems with these techniques were resolved partially by recent work on Long Short-Term Memory networks (Gers and Schmidhuber, 2001), but even this work does not fully do the trick; like more traditional work with recurrent neural nets under gradient descent learning, it shows a tendency to either fall into local optima or take forever to converge on a solution. Hochreiter and Mozer (2001) tried to improve on LSTM networks by abandoning the neural net aspect altogether and proposing a recurrent-net-ish approach based on probability theory; this improved learning in many cases but showed an even worse tendency to fall into local optima — suggesting that a more general probability-based approach is necessary, like the one used in OCP.

Other reinforcement-learning-style methods like Hebbian learning have also failed to scale to very complex problems like the one at hand. Gerald Edelman's (1987) Neural Darwinism approach, which is more a brain theory than a crisp implementable algorithmic formulation, is conceptually somewhat similar to the approach we propose here: he has Hebb-style learning leading to the concretization of *neuronal maps* which are basically neuronal subnetworks that act as encapsulated schema, while still physically overlapping.

And then there is the genetic programming approach, which also has scaling problems, though these can be dealt with more effectively than in the recurrent backprop case. As indicated earlier, we feel that our approach to evolutionary programming resolves GP scaling problems to a significant extent. But we also feel that evolutionary programming is suited only to be part of the solution to procedure learning, not the whole solution.

Finally, there are classifier systems, which will be discussed below (the reinforcement learning aspect of OCP procedure learning bears a significant resemblance to classifier system learning). They share with recurrent backprop some very basic scalability issues. This is why, though these systems have been around some time, there are still no industrial-strength, large-scale applications. Eric Baum's Hayek system (Baum, 2004) solves some of the problems associated with classifier systems, but its learning algorithm is extremely slow, and also highly erratic, requiring substantial customization for each new problem it is applied to.

All in all, we believe that the proposed OCP schema learning approach is conceptually much more advanced than other things that are out there, primarily due to the combination of evolutionary and inferential techniques, and sophistication with which procedure learning is integrated with other cognitive processes.

We are optimistic about the performance of this critical subset of OCP dynamics, though we are also confident that substantial tuning — and substantial processing power — will be required to obtain optimal performance.

Summary Remarks on Procedure Learning

The OCP approach to procedure learning, put as simply as possible, involves

- Evolutionary procedure learning for dealing with *brand new* procedure learning problems, requiring the origination of innovative, highly approximate solutions out of the blue
- Inferential procedure learning for taking approximate solutions and making them exact, and for dealing with procedure learning problems within domains where closely analogous procedure learning problems have previously been solved
- Heuristic, probabilistic data mining for the creation of encapsulated procedures (which then feed into inferential and evolutionary procedure learning)
- PredictiveImplicationLink formation (augmented by PLN inference on such links) as a OCP version of goal-directed reinforcement learning

Using these different learning methods together, as a coherently-tuned whole, one arrives at a holistic procedure learning approach that combines speculation, systematic inference, encapsulation and credit assignment in a single adaptive dynamic process.

It is also worth noting that this hybrid algorithm is highly amenable to distributed processing, even globally distributed processing. Evolutionary programming, probabilistic data mining, and PredictiveImplicationLink formation are extremely well-distributable. Inference is a little less so since it relies on background knowledge at every step; but one can easily envision a procedure learning system that did globally distributed GP and reinforcement learning across millions of widely dispersed machines, and then turned the results of this over to a less widely distributed system carrying out inference.

Predicate Schematization

In this section we treat the process called predicate schematization, by which declarative knowledge about how to carry out actions may be translated into Combo trees embodying specific procedures for carrying out actions. This process is straightforward and automatic in some cases, but in other cases requires significant contextually-savvy inference.

To proceed with predicate schematization, we need the notion of an "executable predicate".

Some predicates are executable in the sense that they correspond to executable schemata, others are not. There are executable atomic predicates (represented by individual PredicateNodes), and executable compound predicates (which are link structures). In general, a predicate may be turned into a schema if it is an atomic executable predicate, or if it is a compound link structure that consists entirely of executable atomic predicates (e.g. pick_up, walk_to, can_do, etc.) and temporal links (e.g. ANDSimultaneous, PredictiveImplication, etc.)

Records of predicate execution may then be made using ExecutionLinks, e.g.

```
ExecutionLink pick_up ( me, ball_7)
```

is a record of the fact that the schema corresponding to the pick_up predicate was executed on the arguments (me, ball_7).

It is also useful to introduce some special (executable) predicates related to schema execution:

- can_do, which represents the system's perceived ability to do something
- do, which denotes the system actually doing something; this is used to mark actions as opposed to perceptions
- just_done, which is true of a schema if the schema has very recently been executed.

The general procedure used in figuring out what predicates to schematize, in order to create a procedure achieving a certain goal, is: Start from the goal and work backwards, following PredictiveImplications and EventualPredictiveImplications and treating can_do's as transparent, stopping when you find something that can currently be done, or else when the process dwindles due to lack of links or lack of sufficiently certain links.

In this process, an ordered list of perceptions and actions will be created. The Atoms in this perception/action-series (PA-series) are linked together via temporal-logical links.

The subtlety of this process, in general, will occur because there may be many different paths to follow. One has the familiar combinatorial explosion of backward-chaining inference, and it may be hard to find the best PA-series among all the mess. Experience-guided pruning is needed here just as with backward-chaining inference.

Specific rules for translating temporal links into executable schemata, used in this process, are as follows. All these rule-statements assume that B is in the selected PA-series. All node variables not preceded by do or can_do are assumed to be perceptions. The ==> symbol denotes the transformation from predicates to executable schemata.

```
EventualPredictiveImplicationLink (do A) B

==>

Repeat (do A) Until B
```

```
EventualPredictiveImplicationLink (do A) (can_do B)

==>
```

```
Repeat
    do A
    do B
Until
    Evaluation just_done B
```

```
// the understanding being that the agent may try to do B and fail,
// and then try again the next time around the loop

PredictiveImplicationLink (do A) (can_do B) <time-lag T>

==>

do A
wait T
do B
```

```
SimultaneousImplicationLink A (can_do B)

==>

if A then do B
```

```
SimultaneousImplicationLink (do A) (can_do B)

==>

do A
do B
```

```
PredictiveImplicationLink A (can_do B)

==>

if A then do B
```

```
SequentialAndLink A1 … An

==>

A1
...
An
```

```
SequentialAndLink A1 … An <time_lag T>

==>
```

```
A1

Wait T

A2

Wait T

...

Wait T

An
```

```
SimultaneousAndLink A1 … An

==>

A1
...
An
```

Note how all instances of can_do are stripped out upon conversion from predicate to schema, and replaced with instances of do.

A Concrete Example

For a specific example of this process, consider the knowledge that: "If I walk to the teacher while whistling, and then give the teacher the ball, I'll get rewarded."

This might be represented by the predicates

```
// walk to the teacher while whistling

A_1 :=

ANDSimultaneous

    do Walk_to

        ExOutLink locate teacher

    EvaluationLink do whistle


// If I walk to the teacher while whistling, eventually I will be next to the teacher

EventualPredictiveImplication
```

```
    A_1
    Evaluation next_to teacher

// While next to the teacher, I can give the teacher the ball
SimultaneousImplication
    EvaluationLink next_to teacher
    can_do
        EvaluationLink give (teacher, ball)

// If I give the teacher the ball, I will get rewarded
PredictiveImplication
    just_done
        EvaluationLink done give (teacher, ball)
    Evaluation reward
```

Via goal-driven predicate schematization, these predicates would become the schemata

```
// walk toward the teacher while whistling
Repeat:
    do WalkTo
        ExOut locate teacher
    do Whistle
Until:
    Evaluation give(teacher, ball)

// if next to the teacher, give the teacher the ball
If:
    Evaluation next_to teacher
```

```
Then

    do give(teacher, ball)
```

Carrying out these two schemata will lead to the desired behavior of walking toward the teacher while whistling, and then giving the teacher the ball when next to the teacher.

Note that, in this example:

- The walk_to, whistle, locate and give used in the example schemata are procedures corresponding to the executable predicates walk_to, whistle, locate and give used in the example predicates
- Next_to is evaluated rather than executed because (unlike the other atomic predicates in the overall predicate being made executable) it has no "do" or "can_do" next to it

(This process was implemented in the Novamente Cognition Engine in 2005, by Ari Heljakka, with details slightly different than those described here, but the essence of the process is the same. This work was reported in a couple papers focused on making NCE play Fetch, one of which is in the proceedings of the 2006 AGI workshop).

Concept-Driven Schema and Predicate Creation

In this section we will deal with the "conversion" of ConceptNodes into SchemaNodes or PredicateNodes. The two cases involve similar but nonidentical methods; we will begin with the simpler PredicateNode case.

Concept-Driven Predicate Creation

Suppose we have a ConceptNode C, with a set of links of the form

```
MemberLink Ai C, i=1,...,n
```

Our goal is to find a PredicateNode so that

```
1)

MemberLink X C

implies

X ''within'' SatisfyingSet(P)

2)

P is as simple as possible
```

This is related to the "Occam's Razor," Solomonoff induction related heuristic to be presented later in this chapter.

We now have an optimization problem: search the space of compound predicates for P that minimize the objective function f(P,C), defined as

```
f(P,C) = w ||P| + (1-w) r(C,P)

r(C,P) =

GetStrength

    EquivalenceLink

        SatisfyingSet( MemberLink X C)

        SatisfyingSet(P)

0 < w < 1 a weight
```

This is an optimization problem over predicate space, which can be solved in an approximate way by the evolutionary programming methods described earlier.

The ConceptPredicatization CIM-Dynamic selects ConceptNodes based on

- OpenCogPrime:Importance
- Total (truth value based) weight of attached OpenCogPrime:MemberLinks and OpenCogPrime:EvaluationLinks

and launches an Evolutionary Learning Task focused on learning predicates based on the nodes it selects. A relevant system parameter here is CPRes, the amount of resources that the system has decided to spend on evolutionary optimization for concept predicatization. This may be expressed, most simply, in terms of a priority in [0,1]. This priority is balanced against the priorities of other processes, and is used by the MultipartAtomspaceController to tell the Evolutionary Optimization Unit-Group how much of its resources to devote to concept predicatization problems. The Evolutionary Optimization Unit Group then allocates or creates a Unit for each new problem that comes its way from one of the instances of the ConceptPredicatization CIM-Dynamic in the system.

Concept-Driven Schema Creation

In the schema learning case, instead of a ConceptNode with MemberLinks and EvaluationLinks, we begin with a ConceptNode C with ExecutionLinks. These ExecutionLinks were presumably produced by inference (the only Novamente cognitive process that knows how to create ExecutionLinks for non-ProcedureNodes).

The optimization problem we have here is: search the space of compound schemata for S that minimize the objective function f(S,C), defined as follows where ||S| refers to the estimated complexity of f (e.g. simply estimated as the size of the normalized Combo tree for S):

```
f(S,C) = w ||S| + (1-w) r(S,P)

Q(S) = the set of pairs (X,Y) so that (ExecutionLink S X Y)

r(S,P) =
```

```
GetStrength

    SubsetLink

        Q(S)

        Graph(S)

0 < w < 1 a weight
```

Operationally, the situation here is very similar to that with concept predicatization. The ConceptSchematization CIM-Dynamic must select ConceptNodes based on:

- OpenCogPrime:Importance
- OpenCogPrime:Total (truth value based) weight of OpenCogPrime:ExecutionLinks

and then feed these to the Evolutionary Optimization Unit-Group, at a rate determined by a system parameter.

Evolution of Pattern-Embodying Predicates

Now we turn to predicate learning — the learning of compound PredicateNodes, in particular.

Aside from logical inference and learning predicates to match existing concepts, how does the system create new predicates? Goal-driven schema learning (via evolution or reinforcement learning) provides an alternate approach: create predicates in the context of creating useful schema. Here we will describe (yet) another complementary dynamic for predicate creation: pattern-oriented PredicateNode evolution.

In most general terms, our approach is to form predicates that embody patterns in itself and in the world. This brings us straight back to the foundations of the patternist philosophy of mind, in which mind is viewed as a system for recognizing patterns in itself and in the world, and then embodying these patterns in itself. This general concept is manifested in many ways in the OCP design, and in this section we will discuss two of them:

- Reward of surprisingly probable Predicates
- Evolutionary learning of pattern-embodying Predicates

These are emphatically not the only way pattern-embodying PredicateNodes get into the system. Inference and concept-based predicate learning also create PredicateNodes embodying patterns. But these two mechanisms complete the picture.

Rewarding Surprising Predicates

The TruthValue of a PredicateNode represents the expected TruthValue obtained by averaging its TruthValue over all its possible legal argument-values. Some Predicates, however, may have high TruthValue without really being *worthwhile.* They may not add any information to their components. We want to identify and reward those Predicates whose TruthValues actually add information beyond what is implicit in the simple fact of combining their components.

For instance, consider the PredicateNode

```
AND

    InheritanceLink X man

    InheritanceLink X ugly
```

If we assume the man and ugly concepts are independent, then this PredicateNode will have the TruthValue

```
man.tv.s * ugly.tv.s
```

What makes the PredicateNode interesting is if:

1. Links it is involved in are important
2. Its TruthValue differs significantly from what would expect based on independence assumptions about its components

It is of value to have interesting Predicates allocated more attention than uninteresting ones. Factor 1 is already taken into account, in a sense: if the PredicateNode is involved in many Links this will boost its activation which will boost its importance. On the other hand, Factor 2 is not taken into account by any previously discussed mechanisms.

It seems to make sense to reward a PredicateNode if it has a surprisingly large or small strength value

One way to do this is to calculate

```
sdiff = ||actual strength – strength predicted via independence assumptions|| *
weight_of_evidence
```

and then increment the value

```
K*sdiff
```

onto the PredicateNode's LongTermImportance value, and similarly increment STI using a different constant.

Another factor that might usefully be caused to increment LTI is the simplicity of a PredicateNode. Given two Predicates with equal strength, we want the system to prefer the simpler one over the more complex one. However, the OckhamsRazor CIM-Dynamic, to be presented above, rewards simpler Predicates directly in their strength values. Hence if the latter is in use, it seems unnecessary to reward them for their simplicity in their LTI values as well. This is an issue that may require some experimentation as the system develops.

Returning to the surprisingness factor, consider the PredicateNode representing

```
AND
     InheritanceLink X cat
     EvaluationLink (eats X) fish
```

If this has a surprisingly high truth value, this means that there are more X known to (or inferred by) the system, that both inherit from *cat* and eat fish, than one would expect given the probabilities of a random X both inheriting from cat and eating fish. Thus, roughly speaking, the conjunction of inheriting from cat and eating fish may be a pattern in the world.

We now see one very clear sense in which OCP dynamics implicitly leads to predicates representing patterns. Small predicates that have surprising truth values get extra activation, hence are more likely to stick around in the system. Thus the mind fills up with patterns.

A More Formal Treatment

It is worth taking a little time to clarify the sense in which we have a *pattern* in the above example, using the mathematical notion of pattern introduced in (Goertzel, 1997) and revised slightly in the Appendix to The Hidden Pattern.

Consider the predicate

```
pred1(T).tv
equals
         >
            GetStrength
                AND
                    Inheritance $X cat
                    Evaluation eats ($X, fish)
            T
```

where T is some threshold value (e.g. .8). Let B = SatisfyingSet(pred1(T)). B is the set of everything that inherits from cat and eats fish.

Now we will make use of the notion of *basic complexity*. If one assumes the entire Atomspace A constituting a given OCP system as given background information, then the basic complexity

```
c(B||A)
```

may be considered as the number of bits required to list the handles of the elements of B, for lookup in A; whereas

```
c(B)
```

is the number of bits required to actually list the elements of B. Now, the formula given above, defining the set B, may be considered as a process P whose output is the set B. The simplicity

```
c(P||A)
```

is the number of bytes needed to describe this process, which is a fairly small number. We assume A is given as background information, accessible to the process.

Then the degree to which P is a pattern in B is given by

```
1 - c(P||A)/c(B||A)
```

which, if B is a sizeable category, is going to be pretty close to 1.

The key to there being a pattern here is that the relation

```
(Inheritance X cat) AND (eats X fish)
```

has a high strength and also a high count. The high count means that B is a large set, either by direct observation or by hypothesis (inference). In the case where the count represents actual pieces of evidence observed by the system and retained in memory, then quite literally and directly, the PredicateNode represents a pattern in a subset of the system (relative to the background knowledge consisting of the system as a whole). On the other hand, if the count value has been obtained indirectly by inference, then it is possible that the system does not actually know any examples of the relation. In this case, the PredicateNode is not a pattern in the actual memory store of the system, but it is being hypothesized to be a pattern in the world in which the system is embedded.

PredicateNode Mining

We have seen how the natural dynamics of the OCP system, with a little help from special heuristics, can lead to the evolution of Predicates that embody patterns in the system's perceived or inferred world. But it is also valuable to more aggressively and directly create pattern-embodying Predicates. This does not contradict the implicit process, but rather complements it. The explicit process we use is called *PredicateNode Mining* and is carried out by a PredicateNodeMiner CIMDynamic.

Define an Atom structure template as a schema expression corresponding to a OCP Link in which some of the arguments are replaced with variables. For instance,

```
Inheritance X cat

EvaluationLink (eats X) fish
```

are Atom structure templates. (Note, as an aside, that Atom structure templates are important in PLN inference control.)

What the PredicateNodeMiner does is to look for Atom structure templates and logical combinations thereof which

- Minimize PredicateNode size
- Maximize surprisingness of truth value

This is accomplished by a combination of heuristics.

The first step in PredicateNode mining is to find Atom structure templates with high truth values. This can be done by a fairly simple heuristic search process.

First, note that if one specifies an (Atom, Link type), one is specifying a set of Atom structure templates. For instance, if one specifies

```
(cat, InheritanceLink)
```

then one is specifying the templates

```
InheritanceLink $X cat
```

and

```
InheritanceLink cat $X
```

One can thus find Atom structure templates as follows. Choose an Atom with high truth value, and then, for each Link type, tabulate the total truth value of the Links of this type involving this Atom. When one finds a promising (Atom, Link type) pair, one can then do inference to test the truth value of the Atom structure template one has found.

Next, given high-truth-value Atom structure templates, the PredicateNodeMiner experiments with joining them together using logical connectives. For each potential combination it assesses the fitness in terms of size and surprisingness. This may be carried out in two ways:

1. By incrementally building up larger combinations from smaller ones, at each incremental stage keeping only those combinations found to be valuable
2. For large combinations, by evolution of combinations

Option 1 is basically greedy data mining (which may be carried out via various standard algorithms), which has the advantage of being much more rapid than evolutionary programming, but the disadvantage that it misses large combinations whose subsets are not as surprising as the combinations themselves. It seems there is room for both approaches in OCP (and potentially many other approaches as well). The PredicateNodeMiner CIM-Dynamic contains a parameter telling it how much time to spend on stochastic pattern mining vs. evolution, as well as parameters guiding the processes it invokes.

So far we have discussed the process of finding single-variable Atom structure templates. But multivariable Atom structure templates may be obtained by combining single-variable ones. For instance, given

```
eats $X fish
```

```
lives_in $X Antarctica
```

one may choose to investigate various combinations such as

```
(eats $X $Y) AND (lives_in $X $Y)
```

(this particular example will have a predictably low truth value). So, the introduction of multiple variables may be done in the same process as the creation of single-variable combinations of Atom structure templates.

When a suitably fit Atom structure template or logical combination thereof is found, then a PredicateNode is created embodying it, and placed into the Atomspace.

Learning Schema Maps

(aka, Reinforcement Learning via HebbianLinks and PredictiveImplicationLinks)

Finally it is time to plunge into the issue of procedure maps -- schema maps in particular. A schema map is a simple yet subtle thing -- a subnetwork of the Atomspace consisting of SchemaNodes, computing some useful quantity or carrying out some useful process in a cooperative way. The general purpose of schema maps is to allow schema execution to interact with other mental processes in a more flexible way than is allowed by compact Combo trees with internal hooks into the Atomspace. Sometimes this isn't necessary -- sometimes a procedure can just be executed with no feedback from other aspect of the mind -- and in these cases a grounded SchemaNode, fully encapsulating the procedure, is the most efficient solution. And sometimes interaction with the rest of the mind is necessary, but can be captured by an appropriate ProcedureEvaluator object. But sometimes execution needs to be even more highly interactive, mediated by attention allocation and other OCP dynamics in a flexible way, and in those cases, one needs a network of ProcedureNodes rather than a single encapsulated ProcedureNode.

But how can schema maps be learned? The basic idea is very simple, and unoriginal: reinforcement learning. Going back at least to Pavlov, the concept couldn't be more natural. In a goal-directed system consisting of interconnected, cooperative elements, you reinforce those connections and/or those elements that have been helpful for achieving goals, and weaken those connections that haven't. Thus, over time, you obtain a network of elements that achieves goals effectively.

In spite of the intuitive appeal of the idea, the long history of reinforcement learning is mostly not a good one. Reinforcement learning has never yielded really good performance on realistic-scale problems, in spite of significant efforts. Of course, few AI methods have ever fulfilled their creators' dreams (yet!), but reinforcement learning has fallen further.

The central difficulty in all reinforcement learning approaches is the 'assignment of credit' problem. If a component of a system has been directly useful for achieving a goal, then rewarding it is easy. But if the relevance of a component to a goal is indirect, then things aren't so simple. Measuring indirect usefulness in a large, richly connected system is difficult -- inaccuracies creep into the process easily, and this, in a nutshell, is why reinforcement learning methods seem to work poorly.

Perhaps the most sophisticated approach to reinforcement learning in any existing AI system is found in John Holland's classifier systems (1992). Classifier systems contain sets of rules, where successful rules are reinforced via a mechanism called 'the bucket brigade' (actually, the first classifier systems used a more complex mechanism, but the bucket brigade approach is the standard in this field now). The bucket brigade algorithm has been shown to be formally equivalent to a variant of reinforcement learning called Q-learning (Sutton and Barto, 1998). Interesting though classifier systems are, though, they have never proved useful in practical applications (unlike Holland's earlier invention, the genetic algorithm). They are phenomenally hard to tune, and don't scale well in the face of complex situations. Baum's Hayek system -- a modification of classifier systems based on principles from theoretical economics -- removes the major problems with the bucket brigade

in theory and in some simple examples, but operates very slowly and appears to have difficulty scaling beyond relatively small 'toy problems.'

There has also been some recent research regarding the use of Bayesian probability theory to carry out reinforcement learning (Sutton and Barto, 1998). However, the work in this vein has suffered from the lack of a probabilistic inference approach capable of dealing with complex networks of events. Complex statistical formulas are proposed that nevertheless only work in special cases far simpler than those confronting actual embodied minds.

[NOTE: the prior paragraphs constitute a brief and inadequate review and could certainly be expanded into a whole page giving an AGI-focused review of reinforcement learning approaches -- Ben G]

We believe that the OCP framework provides the 'support mechanisms and structures' that reinforcement learning needs in order to manifest its potential. We believe that the use of HebbianLinks and PredictiveImplicationLinks for reinforcement learning is much more powerful than approaches that have been taken in the past, precisely because explicit reinforcement learning is not the only process acting on these Links.

Earlier we reviewed the semantics of HebbianLinks, and discussed two methods for forming HebbianLinks:

1. Updating HebbianLink strengths via mining of the System Activity Table
2. Logical inference on HebbianLinks, which may also incorporate the use of inference to combine HebbianLinks with other logical links (for instance, in the reinforcement learning context, PredictiveImplicationLinks)

We will now describe how HebbianLinks, formed and manipulated in this manner, may play a key role in goal-driven reinforcement learning. In effect, what we will describe is an implicit integration of the bucket brigade with PLN inference. The addition of robust probabilistic inference adds a new kind of depth and precision to the reinforcement learning process.

Goal Nodes have an important ability to stimulate a lot of SchemaNode execution activity. If a goal needs to be fulfilled, it stimulates schemata that are known to make this happen. But how is it known which schemata tend to fulfill a given goal? A link

PredictiveImplicationLink S G

means that after schema S has been executed, goal G tends to be fulfilled. If these links between goals and goal-valuable schemata exist, then activation spreading from goals can serve the purpose of causing goal-useful schemata to become active.

The trick, then, is to use HebbianLinks and inference thereon to implicitly guess PredictiveImplicationLinks. A HebbianLink between S1 and S says that when thinking about S1 was useful in the past, thinking about S was also often useful. This suggests that if doing S achieves goal G, maybe doing S1 is also a good idea. The system may then try to find (by direct lookup or reasoning) whether, in the current context, there is a PredictiveImplication joining S1 to S. In this way Hebbian reinforcement learning is being used as an inference control mechanism to aid in the construction of a goal-directed chain of PredictiveImplicationLinks, which may then be schematized into a contextually useful procedure.

Note finally that this process feeds back into itself in an interesting way, via contributing to ongoing HebbianLink formation. Along the way, while leading to the on-the-fly construction of context-appropriate procedures that achieve goals, it also reinforces the HebbianLinks that hold together schema maps, sculpting new schema maps out of the existing field of interlinked SchemaNodes.

Goal-Directed Compound Schema Evolution

Finally, as a complement to goal-driven reinforcement learning, there is also a process of goal-directed compound SchemaNode evolution. This combines features of the goal-driven reinforcement learning and concept-driven schema evolution methods discussed above. Here we use a Goal Node to provide the fitness function for schema evolution.

The basic idea is that the fitness of a schema is defined by the degree to which enaction of that schema causes fulfillment of the goal. This requires the introduction of CausalImplicationLinks, as defined in Probabilistic Logic Networks. In the simplest case, a CausalImplicationLink is simply a PredictiveImplicationLink.

One relatively simple implementation of the idea is as follows. Suppose we have a Goal Node G, whose satisfaction we desire to have achieved by time T1. Suppose we want to find a SchemaNode S whose execution at time T2 will cause G to be achieved. We may define a fitness function for evaluating candidate S by

```
f(S,G,T1,T2) = w * r(S,P,T1,T2) + (1-w) size(S)

0 < w < 1 a weight, e.g. w=.5

r(S,P,T1,T2) =

      GetStrength

            CausalImplicationLink

                  EvaluationLink

                        AtTime

                              T1

                              ExecutionLink S * *

                  EvaluationLink AtTime (T2, G)
```

Another variant specifies only a relative time lag, not two absolute times.

```
f(S,G,T) = w * v(S,P,T) + (1-w) size(S)

0 < w < 1 a weight, e.g. w=.5
```

```
v(S,P,T) =

      AND

            NonEmpty

                  SatisfyingSet r(S,P,T1,T2)

            T1 > T2 - T
```

Using evolutionary learning to find schemata fulfilling these fitness functions, results in compound SchemaNodes whose execution is expected to cause the achievement of given goals. This is a complementary approach to reinforcement-learning based schema learning, and to schema learning based on PredicateNode concept creation. The strengths and weaknesses of these different approaches need to be extensively experimentally explored. However, prior experience with the learning algorithms involved gives us some guidance.

We know that when absolutely nothing is known about an objective function, evolutionary programming is often the best way to proceed. Even when there is knowledge about an objective function, the evolution process can take it into account, because the fitness functions involve logical links, and the evaluation of these logical links may involve inference operations.

On the other hand, when there's a lot of relevant knowledge embodied in previously executed procedures, using logical reasoning to guide new procedure creation can be cumbersome. The Hebbian mechanisms used in reinforcement learning may be understood as inferential in their conceptual foundations (since a HebbianLink is equivalent to an ImplicationLink between two propositions about importance levels). But in practice they provide a much-streamlined approach to bringing knowledge implicit in existing procedures to bear on the creation of new procedures. Reinforcement learning, we believe, will excel at combining existing procedures to form new ones, and modifying existing procedures to work well in new contexts. Logical inference can also help here, acting in cooperation with reinforcement learning. But when the system has no clue how a certain goal might be fulfilled, evolutionary schema learning provides a relatively time-efficient way for it to find something minimally workable.

Pragmatically, the GoalDrivenSchemaLearning CIM-Dynamic handles this aspect of the system's operations. It selects Goal Nodes with probability proportional to importance, and then spawns problems for the Evolutionary Optimization Unit Group accordingly. For a given Goal Node, PLN control mechanisms are used to study its properties and select between the above objective functions to use, on an heuristic basis.

Occam's Razor

This brief pages discusses an important cognitive process that fits loosely into the category of "OCP Procedure learning" — it's not actually a *procedure learning* process, but rather a process that utilizes the fruits of procedure learning.

The well-known "Occam's razor" heuristic says that all else equal, simpler is better. This notion is embodied mathematically in the Solomonoff-Levin "universal prior," according to which the a priori probability of a computational entity X is defined as a normalized version of

```
m(X) = Sum_p 2^(-l(p))
```

where

- the sum is taken over all programs p that compute X
- l(p) denotes the length of the program p

Normalization is necessary because these values will not automatically sum to 1 over the space of all X.

Without normalization, m is a semimeasure rather than a measure; with normalization it becomes the "Solomonoff-Levin measure" (Zvonkin and Levin, 1970).

Roughly speaking, Solomonoff's induction theorem (Solomonoff, 1964, 1978) shows that, if one is trying to learn the computer program underlying a given set of observed data, and one does Bayesian inference over the set of all programs to try and obtain the answer, then if one uses the universal prior distribution one will arrive at the correct answer.

Novamente is not a Solomonoff induction engine. The computational cost of actually applying Solomonoff induction is unrealistically large. However, as we have seen in this chapter, there are aspects of Novamente that are reminiscent of Solomonoff induction. In concept-directed schema and predicate learning, in pattern-based predicate learning — and in causal schema learning, we are searching for schemata and predicates that minimize complexity while maximizing some other quality. These processes all implement the Occam's Razor heuristic in a Solomonoffian style.

Now we will introduce one more method of imposing the heuristic of algorithmic simplicity on Novamente Atoms (and hence, indirectly, on Novamente maps as well). This is simply to give a higher a priori probability to entities that are more simply computable.

For starters, we may increase the node probability of ProcedureNodes proportionately to their simplicity. A reasonable formula here is simply

```
2 ^( -r c(P) )
```

where P is the ProcedureNode and r>0 is a parameter. This means that infinitely complex P have a priori probability zero, whereas an infinitely simple P has an a priori probability 1.

This is not an exact implementation of the Solomonoff-Levin measure, but it's a decent heuristic approximation. It is not pragmatically realistic to sum over the lengths of all programs that do the same thing as a given predicate P. Generally the first term of the Solomonoff-Levin summation is going to dominate the sum anyway, so if the ProcedureNode P is maximally compact, then our simplified formula will be a good approximation of the Solomonoff-Levin summation.

These a priori probabilities may be merged with node probability estimates from other sources, using the revision rule. The question arises how much weight of evidence to assign to the a priori probability as opposed to other sources. This again becomes a system parameter.[BNG1]

A similar strategy may be taken with ConceptNodes. We want to reward a ConceptNode C with a higher a priori probability if C » SatisfyingSet(P) for a simple PredicateNode P. To achieve this formulaically, let sim(X,Y) denote the strength of the SimilarityLink between X and Y, and let

```
sim*(C,P) = sim( C, SatisfyingSet(P) )
```

We may then define the a priori probability of a ConceptNode as

```
pr(C) = Sum_P sim*(C,P) 2^(-r c(P))
```

where the sum goes over all P in the system. In practice of course it's only necessary to compute the terms of the sum corresponding to P so that sim*(C,P) is large.

As with the a priori PredicateNode probabilities discussed above, these a priori ConceptNode probabilities may be merged with other node probability information, using the revision rule, and using a default parameter value for the weight of evidence. There is one pragmatic difference here from the PredicateNode case, though. As the system learns new PredicateNodes, its best estimate of pr(C) may change. Thus it makes sense for the system to store the a priori probabilities of ConceptNodes separately from the node probabilities, so that when the a priori probability is changed, a two step operation can be carried out:

- First, remove the old a priori probability from the node probability estimate, using the reverse of the revision rule
- Then, add in the new a priori probability

Finally, we can take a similar approach to any Atom Y produced by a SchemaNode. We can construct

```
pr(Y) = Sum_{S,X} [ s(S,X,Y) 2^(-r (c(S)+c(X))) ]
```

where the sum goes over all pairs (S,X) so that

```
ExecutionLink S X Y
```

and s(S,X,Y) is the strength of this ExecutionLink. Here, we are rewarding Atoms that are produced by simple schemata based on simple inputs.

The combined result of these heuristics is to cause the system to prefer simpler explanations, analyses, procedures and ideas. But of course this is only an a priori preference, and if more complex entities prove more useful, these will quickly gain greater strength and importance in the system.

Implementationally, these various processes are carried out by the OccamsRazor CIMDynamic. This dynamic selects ConceptNodes based on a combination of

- importance
- time since the a priori probability was last updated (a long time is preferred)

It selects ExecutionLinks based on importance and based on the amount of time since they were last visited by the OccamsRazor CIMDynamic. And it selects PredicateNodes based on importance, filtering out PredicateNodes it has visited before.

Chapter Sixteen: Map Encapsulation and Expansion

OpenCogPrime:MapEncapsulationAndExpansion

- OpenCogPrime:ActivityTables
- OpenCogPrime:MapMining
- OpenCogPrime:ProcedureEncapsulation
- OpenCogPrime:MapsAndAttention

In most of these wiki pages, we have focused primarily on the Atom level of representation and dynamics, commenting on the map level now and then but primarily leaving it implicit. There are two reasons for this: one theoretical, one pragmatic. The theoretical reason is that the majority of map dynamics and representations are implicit in Atom-level correlates. And the pragmatic reason is that, at this stage, we simply do not know as much about OCP maps as we do about OCP Atoms. Maps are emergent entities and, lacking a detailed theory of OCP dynamics, the only way we have to study them in detail is to run OCP systems and mine their System Activity Tables for information. If research goes well, then later versions of this wiki, revised to account for more extensive experimentation with more complete OCP versions, may include more details on observed map dynamics in various contexts.

In this page and the ones it supervenes over, however, we will briefly turn our gaze explicitly to maps and their relationships to Atoms, and discuss processes that convert Atoms into maps (expansion) and vice versa (encapsulation). These processes represent a bridge between the concretely-implemented and emergent aspects of OCP's mind.

In SMEPH terms, the encapsulation process is how OCP explicitly studies its own derived hypergraph and then works to implement this derived hypergraph more efficiently by recapitulating it at the concretely-implemented-mind level. This of course may change the derived hypergraph considerably. Among other things, it has the possibility of taking the things that were the most abstract, *highest level* patterns in the system and forming new patterns involving them and their interrelationships — thus building the highest level of patterns in the system higher and higher. Expansion, on the other hand, is critical in the case of ProcedureNodes, because it involves taking their internal Combo trees and expanding them into distributed processes involving multiple Nodes and Links — a process that, when enacted on a procedure P, enables PLN, attention allocation and other cognitive processes to work on P effectively, as well as increases the odds that P will occupy enough of the system's knowledge and resources to get a significant role in its derived hypergraph.

As of July 2008, this chapter is mainly prospective, describing plans for future implementation rather than existing code and work already done. Map expansion has been coded, but it is fairly trivial software-wise; the bigger job of map encapsulation still remains untouched in the July 2008 codebase. However, it won't require a lot of new implementation either: it will mainly be a matter of tuning, testing and generally experimenting. Implementation-wise, it *just* requires the creation of some special tables, and code for dynamically maintaining them, and then the application of other (already-described) cognitive processes to the problem of pattern-mining

on them. The elegant side of the OCP design shows up here, in terms of the reuse of the same basic cognitive processes for multiple qualitatively different purposes.

In spite of their relative software simplicity in a OCP context, however, the philosophical essentialness of these issues should not be underestimated. As noted above, Atoms created via map encapsulation may then become involved in new maps, which may then be recognized and turned into Atoms. This process of map encapsulation is an important OCP manifestation of the general psynet-model principle that mind is a set of actors that create new actors embodying patterns recognized in and amongst one another. A map is a pattern recognizable amongst a set of Atoms over time; when this pattern is recognized and embodied as an Atom, the *emergence* of the Atomspace is passing over into the *concrete material* making up the Atomspace. According to the psynet model, any intelligent system is going to manifest this sort of iterative passage from emergence to concreteness somehow. How the brain does it, we really do not yet know. What we have done is to outline how such a thing may happen in the particular context of OCP. How well it will work in practice remains to be seen of course; but so far as we know, no other AI design has seriously addressed such issues.

The Significance of Map Encapsulation and Expansion

Let's start with the less interesting case: map expansion. Converting Atoms into maps is something that normally occurs only implicitly. For instance, a ConceptNode C may turn into a concept map if the importance updating process iteratively acts in such a way as to create/reinforce a map consisting of C and its relata. And we will shortly review another example, where an Atom-level InheritanceLink may implicitly spawn a map-level InheritanceEdge. However, there is one important case in which Atom-to-map conversion must occur explicitly: the expansion of compound ProcedureNodes into procedure maps. This must occur explicitly because the process graphs inside ProcedureNodes have no dynamics going on except evaluation; there is no opportunity for them to manifest themselves as maps, unless a MindAgent is introduced that explicitly does so. Of course, just unfolding a Combo tree into a procedure map doesn't intrinsically make it a significant part of the derived hypergraph —— but it opens the door for the inter-cognitive-process integration that may make this occur.

Next, encapsulation may be viewed as a form of *symbolization,* in which the system creates concrete entities to serve as symbols for its own emergent patterns. It can then study an emergent pattern's interrelationships by studying the interrelationships of the symbol with other symbols. For instance, suppose one has three derived-hypergraph ConceptVertices A, B and C, and observes that

```
InheritanceEdge A B

InheritanceEdge B C
```

Then encapsulation may create ConceptNodes A', B' and C' for A, B and C, and InheritanceLinks corresponding to the InheritanceEdges, where e.g. A' is a set containing all the Atoms contained in the static map A. First-order PLN inference will then immediately conclude

```
InheritanceLink A' C'
```

and it may possibly do so with a higher strength than the strength corresponding to the (perhaps not significant)

InheritanceEdge between A and C. But if the encapsulation is done right then the existence of the new InheritanceLink will indirectly cause the formation of the corresponding

```
InheritanceEdge A C
```

via the further action of inference, which will use (InheritanceLink A' C') to trigger the inference of further inheritance relationships between members of A' and members of C', which will create an emergent inheritance between members of A (the map corresponding to A') and C (the map corresponding to C').

The above example involved the conversion of static maps into ConceptNodes. Another approach to map encapsulation is to represent the fact that a set of Atoms constitutes a map as a predicate; for instance if the nodes A, B and C are habitually used together, then the predicate P may be formed, where

```
P =

AND

          A is used at time T

          B is used at time T

          C is used at time T
```

The habitualness of A, B and C being used together will be reflected in the fact that P has a surprisingly high truth value. By a simple concept formation heuristic, this may be used to form a link AND(A, B, C), so that

```
AND(A, B, C) is used at time T
```

This composite link AND(A, B, C) is then an embodiment of the map in single-Atom form.

Similarly, if a set of schemata is commonly used in a certain series, this may be recognized in a predicate, and a composite schema may then be created embodying the component schemata. For instance, suppose it is recognized as a pattern that

```
AND

                S1 is used at time T on input I1 producing output O1

                S2 is used at time T+s on input O1 producing output O2
```

Then we may explicitly create a schema that consists of S1 taking input and feeding its output to S2. This cannot be done via any standard concept formation heuristic; it requires a special process.

One might wonder why this Atom-to-map conversion process is necessary: Why not just let maps combine to build new maps, hierarchically, rather than artificially transforming some maps into Atoms and letting maps then form from these map-representing Atoms. It is all a matter of precision. Operations on the map level are fuzzier and less reliable than operations on the Atom level. This fuzziness has its positive and its negative aspects. For example, it is good for spontaneous creativity, but bad for constructing lengthy, confident chains of thought.

OpenCogPrime:MapEncapsulationAndExpansion

- OpenCogPrime:ActivityTables
- OpenCogPrime:MapMining
- OpenCogPrime:ProcedureEncapsulation
- OpenCogPrime:MapsAndAttention

In most of these wiki pages, we have focused primarily on the Atom level of representation and dynamics, commenting on the map level now and then but primarily leaving it implicit. There are two reasons for this: one theoretical, one pragmatic. The theoretical reason is that the majority of map dynamics and representations are implicit in Atom-level correlates. And the pragmatic reason is that, at this stage, we simply do not know as much about OCP maps as we do about OCP Atoms. Maps are emergent entities and, lacking a detailed theory of OCP dynamics, the only way we have to study them in detail is to run OCP systems and mine their System Activity Tables for information. If research goes well, then later versions of this wiki, revised to account for more extensive experimentation with more complete OCP versions, may include more details on observed map dynamics in various contexts.

In this page and the ones it supervenes over, however, we will briefly turn our gaze explicitly to maps and their relationships to Atoms, and discuss processes that convert Atoms into maps (expansion) and vice versa (encapsulation). These processes represent a bridge between the concretely-implemented and emergent aspects of OCP's mind.

In SMEPH terms, the encapsulation process is how OCP explicitly studies its own derived hypergraph and then works to implement this derived hypergraph more efficiently by recapitulating it at the concretely-implemented-mind level. This of course may change the derived hypergraph considerably. Among other things, it has the possibility of taking the things that were the most abstract, *highest level* patterns in the system and forming new patterns involving them and their interrelationships — thus building the highest level of patterns in the system higher and higher. Expansion, on the other hand, is critical in the case of ProcedureNodes, because it involves taking their internal Combo trees and expanding them into distributed processes involving multiple Nodes and Links — a process that, when enacted on a procedure P, enables PLN, attention allocation and other cognitive processes to work on P effectively, as well as increases the odds that P will occupy enough of the system's knowledge and resources to get a significant role in its derived hypergraph.

As of July 2008, this chapter is mainly prospective, describing plans for future implementation rather than existing code and work already done. Map expansion has been coded, but it is fairly trivial software-wise; the bigger job of map encapsulation still remains untouched in the July 2008 codebase. However, it won't require a lot of new implementation either: it will mainly be a matter of tuning, testing and generally experimenting. Implementation-wise, it *just* requires the creation of some special tables, and code for dynamically maintaining them, and then the application of other (already-described) cognitive processes to the problem of pattern-mining on them. The elegant side of the OCP design shows up here, in terms of the reuse of the same basic cognitive processes for multiple qualitatively different purposes.

In spite of their relative software simplicity in a OCP context, however, the philosophical essentialness of these issues should not be underestimated. As noted above, Atoms created via map encapsulation may then become involved in new maps, which may then be recognized and turned into Atoms. This process of map encapsulation is an important OCP manifestation of the general psynet-model principle that mind is a set of actors that create new actors embodying patterns recognized in and amongst one another. A map is a pattern

recognizable amongst a set of Atoms over time; when this pattern is recognized and embodied as an Atom, the *emergence* of the Atomspace is passing over into the *concrete material* making up the Atomspace. According to the psynet model, any intelligent system is going to manifest this sort of iterative passage from emergence to concreteness somehow. How the brain does it, we really do not yet know. What we have done is to outline how such a thing may happen in the particular context of OCP. How well it will work in practice remains to be seen of course; but so far as we know, no other AI design has seriously addressed such issues.

The Significance of Map Encapsulation and Expansion

Let's start with the less interesting case: map expansion. Converting Atoms into maps is something that normally occurs only implicitly. For instance, a ConceptNode C may turn into a concept map if the importance updating process iteratively acts in such a way as to create/reinforce a map consisting of C and its relata. And we will shortly review another example, where an Atom-level InheritanceLink may implicitly spawn a map-level InheritanceEdge. However, there is one important case in which Atom-to-map conversion must occur explicitly: the expansion of compound ProcedureNodes into procedure maps. This must occur explicitly because the process graphs inside ProcedureNodes have no dynamics going on except evaluation; there is no opportunity for them to manifest themselves as maps, unless a MindAgent is introduced that explicitly does so. Of course, just unfolding a Combo tree into a procedure map doesn't intrinsically make it a significant part of the derived hypergraph —— but it opens the door for the inter-cognitive-process integration that may make this occur.

Next, encapsulation may be viewed as a form of *symbolization,* in which the system creates concrete entities to serve as symbols for its own emergent patterns. It can then study an emergent pattern's interrelationships by studying the interrelationships of the symbol with other symbols. For instance, suppose one has three derived-hypergraph ConceptVertices A, B and C, and observes that

```
InheritanceEdge A B

InheritanceEdge B C
```

Then encapsulation may create ConceptNodes A', B' and C' for A, B and C, and InheritanceLinks corresponding to the InheritanceEdges, where e.g. A' is a set containing all the Atoms contained in the static map A. First-order PLN inference will then immediately conclude

```
InheritanceLink A' C'
```

and it may possibly do so with a higher strength than the strength corresponding to the (perhaps not significant) InheritanceEdge between A and C. But if the encapsulation is done right then the existence of the new InheritanceLink will indirectly cause the formation of the corresponding

```
InheritanceEdge A C
```

via the further action of inference, which will use (InheritanceLink A' C') to trigger the inference of further inheritance relationships between members of A' and members of C', which will create an emergent inheritance between members of A (the map corresponding to A') and C (the map corresponding to C').

The above example involved the conversion of static maps into ConceptNodes. Another approach to map encapsulation is to represent the fact that a set of Atoms constitutes a map as a predicate; for instance if the nodes A, B and C are habitually used together, then the predicate P may be formed, where

```
P =

AND

          A is used at time T

          B is used at time T

          C is used at time T
```

The habitualness of A, B and C being used together will be reflected in the fact that P has a surprisingly high truth value. By a simple concept formation heuristic, this may be used to form a link AND(A, B, C), so that

```
AND(A, B, C) is used at time T
```

This composite link AND(A, B, C) is then an embodiment of the map in single-Atom form.

Similarly, if a set of schemata is commonly used in a certain series, this may be recognized in a predicate, and a composite schema may then be created embodying the component schemata. For instance, suppose it is recognized as a pattern that

```
AND

                S1 is used at time T on input I1 producing output O1

                S2 is used at time T+s on input O1 producing output O2
```

Then we may explicitly create a schema that consists of S1 taking input and feeding its output to S2. This cannot be done via any standard concept formation heuristic; it requires a special process.

One might wonder why this Atom-to-map conversion process is necessary: Why not just let maps combine to build new maps, hierarchically, rather than artificially transforming some maps into Atoms and letting maps then form from these map-representing Atoms. It is all a matter of precision. Operations on the map level are fuzzier and less reliable than operations on the Atom level. This fuzziness has its positive and its negative aspects. For example, it is good for spontaneous creativity, but bad for constructing lengthy, confident chains of thought.

Mining the Atomspace for Maps

Searching for general maps in a complex Atomspace is an unrealistically difficult problem, as the search space is huge. So, the bulk of map-mining activity involves looking for the most simple and obvious sorts of maps. A certain amount of resources may also be allocated to looking for subtler maps using more resource-intensive methods.

The following categories of maps can be searched for at relatively low cost:

- Static maps
- Temporal motif maps

Conceptually, a static map is simply a set of Atoms that all tend to be active at the same time.

Next, by a "temporal motif map" we mean a set of pairs

```
(Ai,ti)
```

of the type

```
(Atom, int)
```

so that for many activation cycle indices T, Ai is highly active at some time very close to index T + ti

The reason both static maps and temporal motif maps are easy to recognize is that they are both simply repeated patterns.

Perceptual context formation involves a special case of static and temporal motif mining.In perceptual context formation, one specifically wishes to mine maps involving perceptual nodes associated with a single interaction channel. These maps then represent real-world contexts, that may be useful in guiding real-world-oriented goal activity (via schema-context-goal triads).

We have three approaches for mining static and temporal motif maps from Novamente Atomspaces:

- Frequent subgraph mining, frequent itemset mining, or other sorts of datamining on Activity Tables
- Clustering on the network of HebbianLinks
- Evolutionary Optimization based datamining on Activity Tables

The first two approaches are significantly more time-efficient than the latter, but also significantly more limited in the scope of patterns they can find.

Any of these approaches can be used to look for maps subject to several types of constraints:

- Unconstrained: maps may contain any kinds of Atoms
- Strictly constrained: maps may only contain Atom types contained on a certain list
- Probabilistically constrained: maps must contain Atom types contained on a certain list, as x% of their elements
- Trigger-constrained: the map must contain an Atom whose type is on a certain list, as its most active element

Different sorts of constraints will lead to different sorts of maps, of course. We don't know at this stage which sorts of constraints will yield the best results. Some special cases, however, are reasonably well understood. For instance

- procedure encapsulation, to be discussed below, involves searching for (strictly-constrained) maps consisting solely of ProcedureInstanceNodes.
- to enhance goal achievement, it is likely useful to search for trigger-constrained maps triggered by Goal Nodes

What the MapEncapsulation CIM-Dynamic does once it finds a map, is dependent upon the type of map it's found.

In the special case of procedure encapsulation, it creates a compound ProcedureNode (selecting SchemaNode or PredicateNode based on whether the output is a TruthValue or not).

For static maps, it creates a ConceptNode, which links to all members of the map with InheritanceLinks, the weight of which is determined by the degree of map membership.

Map Encapsulation Using Frequent Itemset or Subgraph Mining

In principle one could use evolutionary learning to do all Novamente's map encapsulation, but this isn't computationally feasible — it would limit too severely the amount of map encapsulation that could be done. Instead, evolutionary learning must be supplemented by some more rapid, less expensive technique.

Frequent Itemset Mining for Map Mining

One class of technique that is useful here is *frequent itemset mining* (FIM), a process that looks to find all frequent combinations of items occuring in a set of data. There are a number of algorithms in this category, the classic is Apriori (Agrawal and Srikant, 1994) but we have recently been experimenting with an alternative called relim (Borgelt, 2005) which is conceptually similar but seems to give better performance. Another useful class of algorithms is greedy or stochastic itemset mining, which does roughly the same thing as FIM but without being completely exhaustive (the advantage being greater execution speed). Here we will discuss FIM, but the basic concepts are the same if one is doing greedy or stochastic mining instead.

The basic goal of frequent itemset mining is to discover frequent subsets in a group of items. One knows that for a set of N items, there are 2N-1 possible subgroups. To avoid the exponential explosion of subsets, one may compute the frequent *itemsets* in several rounds. Round i computes all frequent i-itemsets.

A round has two steps: candidate generation and candidate counting. In the candidate generation step, the algorithm generates a set of candidate i-itemsets whose support — a minimum percentage of events in which the item must appear — has not been yet been computed. In the candidate-counting step, the algorithm scans its memory database, counting the support of the candidate itemsets. After the scan, the algorithm discards candidates with support lower than the specified minimum (an algorithm parameter) and retains only the frequent i-itemsets. The algorithm reduces the number of tested subsets by pruning a priori those candidate itemsets that cannot be frequent, based on the knowledge about infrequent itemsets obtained from previous rounds. Although the worst case of this sort of algorithm is exponential, practical executions are generally fast, depending essentially on the support limit.

To apply this kind of approach to search for static maps, one simply creates a large set of sets of Atoms — one set for each time-point. In the set S(t) corresponding to time t, we place all Atoms that were firing activation at time t. The itemset miner then searches for sets of Atoms that are subsets of many different S(t) corresponding to many different times t. These are Atom sets that are frequently co-active.

Table 1 (see end of page for attachment) presents a typical example of data prepared for frequent itemset mining, in the context of context formation via static-map recognition. Columns represent interaction-channel-

important nodes and rows indicate time slices. For simplicity, we have *thresholded* the values and show only activity values; so that a 1 in a cell indicates that the Atom indicated by the column was being utilized at the time indicated by the row.

In the example, if we assume minimum support as 50 percent, the context nodes C1 = {Q, R}, and C2 = {Q, T, U} would be created.

Using frequent itemset mining to find temporal motif maps is a similar, but slightly more complex process. Here, one fixes a time-window W. Then, for each activation cycle index t, one creates a set S(t) consisting of pairs of the form

```
(A, s)
```

where A is an Atom and 0<=s<=W is an integer temporal offset. We have

```
(A,s) ''within'' S(t)
```

if Atom A is firing activation at time t+s.Itemset mining is then used to search for common subsets among the S(t). These common subsets are common patterns of temporal activation, i.e. repeated temporal motifs.

The strength of this approach is its ability to rapidly search through a huge space of possibly significant subsets. Its weakness is its restriction to finding maps that can be incrementally built up from smaller maps.How significant this weakness is, depends on the particular statistics of map occurrence in Novamente. Intuitively, we believe frequent itemset mining can perform rather well in this context, and our preliminary experiments have supported this intuition.

Frequent Subgraph Mining for Map Mining

A limitation of FIM techniques, from an OCP perspective, is that they are intended for relational databases; but the information about co-activity in an OCP instance is generally going to be more efficiently stored as graphs rather than RDB's. Indeed an ActivityTable may be effectively stored as a series of graphs corresponding to time intervals — one graph for each interval, consisting of HebbianLinks formed solely based on importance during that interval. From an ActivityTable stores like this, the way to find maps is not frequent itemset mining but rather frequent subgraph mining, a variant of FIM that is conceptually similar but algorithmically more subtle, and on which there has arisen a significant literature in recent years.

Evolutionary Map Detection

Complementarily to the itemset mining approach, the Novamente design also uses evolutionary optimization to find maps. Here the data setup is the same as in the itemset mining case, but instead of using an incremental search approach, one sets up a population of subsets of the sets S(t), and seeks to evolve the population to find an optimally fit S(t). Fitness is defined simply as high frequency - relative to the frequency one would expect based on statistical independence assumptions alone.

In this context, one probably wishes to use a variant of evolutionary optimization using some diversity preservation mechanism, so that the population contains not merely a single most optimal subset, but a number

of unrelated subsets, all of which are reasonably fit. EDA's like MOSES are fairly good in this regard, having much less tendency than GP to converge to highly homogenous final populations.

Map Dynamics

```
Assume one has a collection of Atoms, with
```

- Importance values I(A), assigned via the economic attention allocation mechanism
- HebbianLink strengths (HebbianLink A B).s, assigned as (loosely speaking) the probability of B's importance assuming A's importance

Then, one way to search for static maps is to look for collections C of Atoms that are strong clusters according to HebbianLinks. That is, for instance, to find collections C so that

```
The mean strength of (HebbianLink A B).s, where A and B are in the collection C >>
The mean strength of (HebbianLink A Z).s, where A is in the collection C and Z is not
```

(this is just a very simple cluster quality measurement; there is a variety of other cluster quality measurements one might use instead.)

Dynamic maps may be more complex, for instance there might be two collections C1 and C2 so that

```
Mean strength of (HebbianLink A B).s, where A is in C1 and B is in C2

Mean strength of (HebbianLink B A).s, where B is in C2 and A is in C1
```

are both very large.

A static map will tend to be an attractor for OCP's attention-allocation-based dynamics, in the sense that when a few elements of the map are acted upon, it is likely that other elements of the map will soon also come to be acted upon. The reason is that, if a few elements of the map are acted upon usefully, then their importance values will increase. Node probability inference based on the HebbianLinks will then cause the importance values of the other nodes in the map to increase, thus increasing the probability that the other nodes in the map are acted upon. Critical here is that the HebbianLinks have a higher weight of evidence than the node importance values. This is because the node importance values are assumed to be ephemeral — they reflect whether a given node is important at a given moment or not — whereas the HebbianLinks are assumed to reflect longer-lasting information.

A dynamic map will also be an attractor, but of a more complex kind. The example given above, with C1 and C2, will be a periodic attractor rather than a fixed-point attractor.

Procedure Encapsulation and Expansion

One of the most important special cases of map encapsulation is procedure encapsulation. This refers to the process of taking a schema/predicate map and embodying it in a single ProcedureNode.This may be done by mining on the Procedure Activity Table, described in Activity Tables, using either

- a special variant of itemset mining that seeks for procedures whose outputs serve as inputs for other procedures.
- Evolutionary optimization with a fitness function that restricts attention to sets of procedures that form a digraph, where the procedures lie at the vertices and an arrow from vertex A to vertex B indicates that the outputs of A become the inputs of B

The reverse of this process, procedure expansion, is also interesting, though algorithmically easier-- here one takes a compound ProcedureNode and expands its internals into a collection of appropriately interlinked ProcedureNodes. The challenge here is to figure out where to split a complex Combo tree into subtrees. But if the Combo tree has a hierarchical structure then this is very simple; the hierarchical subunits may simply be split into separate ProcedureNodes.

These two processes may be used in sequence to interesting effect: expanding an important compound ProcedureNode so it can be modified via reinforcement learning, then encapsulating its modified version for efficient execution, then perhaps expanding this modified version later on.

To an extent, the existence of these two different representations of procedures is an artifact of OCP's particular software design (and ultimately, a reflection of certain properties of the von Neumann computing architecture). But it also represents a more fundamental dichotomy, between

- Procedures represented in a way that is open to interaction with other mental processes during the execution process
- Procedures represented in a way that is encapsulated and mechanical, with no room for interference from other aspects of the mind during execution

Conceptually, we believe that this is a very useful distinction for a mind to make. In nearly any reasonable cognitive architecture, it's going to be more efficient to execute a procedure if that procedure is treated as a world unto itself, so it can simply be executed without worrying about interactions. This is a strong motivation for an artificial cognitive system to have a dual (at least) representation of procedures.

Procedure Encapsulation in More Detail

For a visual depiction of the procedure encapsulation process, see the attachment.

A procedure map is a temporal motif: it is a set of Atoms (ProcedureNodes), which are habitually executed in a particular temporal order, and which implicitly pass arguments amongst each other. For instance, if procedure A acts to create node X, and procedure B then takes node X as input, then we may say that A has implicitly passed an argument to B.

The encapsulation process can recognize some very subtle patterns, but a fair fraction of its activity can be understood in terms of some simple heuristics.

For instance, map encapsulation process will create a node h = B f g = *f composed with g* when there are many examples in the system of

```
ExecutionLink g x y

ExecutionLink f y z
```

The procedure encapsulation process will also recognize larger repeated subgraphs, and their patterns of execution over time. But some of its recognition of larger subgraphs may be done incrementally, by repeated recognition of simple patterns like the ones just described.

Procedure Encapsulation in the Human Brain

Finally, we will briefly discuss some conceptual issues regarding the relation between OCP procedure encapsulation and the human brain. Current knowledge of the human brain is weak in this regard, but we won't be surprised if, in time, it is revealed that the brain stores procedures in several different ways, that one distinction between these different ways has to do with degree of openness to interactions, and that the less open ways lead to faster execution.

Generally speaking, there is good evidence for a neural distinction between procedural, episodic and declarative memory. But knowledge about distinctions between different kinds of procedural memory is scantier. It is known that procedural knowledge can be "routinized" — so that, e.g., once you get good at serving a tennis ball or solving a quadratic equation, your brain handles the process in a different way than before when you were learning. And it seems plausible that routinized knowledge, as represented in the brain, has fewer connections back to the rest of the brain than the pre-routinized knowledge. But there will be much firmer knowledge about such things in 5-10 years time as brain scanning technology advances.

There is more neuroscience knowledge about motor procedures than cognitive procedures (see e.g. Reza and Wise, 2005). Much of motor procedural memory resides in the pre-motor area of the cortex. The *motor plans* stored here are not static entities and are easily modified through feedback, and through interaction with other brain regions. Generally, a motor plan will be stored in a distributed way across a significant percentage of the premotor cortex; and a complex or multipart actions will tend to involve numerous sub-plans, executed in both parallel and in serial. Often what we think of as separate/distinct motor-plans may in fact be just slightly different combinations of subplans (a phenomenon also occurring with schema maps in OCP).

In the case of motor plans, a great deal of the *routinization* process has to do with learning the timing necessary for correct coordination between muscles and motor subplans. This involves integration of several brain regions — for instance, timing is handled by the cerebellum to a degree, and some motor-execution decisions are regulated by the basal ganglia.

One can think of many motor plans as involving abstract and concrete sub-plans. The abstract sub-plans are more likely to involve integration with those parts of the cortex dealing with conceptual thought. The concrete sub-plans have highly optimized timings, based on close integration with cerebellum, basal ganglia and so forth — but are not closely integrated with the conceptualization-focused parts of the brain. So, a rough OCP model of human motor procedures might involve schema maps coordinating the abstract aspects of motor procedures, triggering activity of complex SchemaNodes containing precisely optimized procedures that interact carefully with external actuators.

Maps and Focused Attention

The cause of map formation is important to understand. Interestingly, small maps seems to form by the logic of focused attention, as well as hierarchical maps of a certain nature. A few more comments on this may be of use.

The nature of PLN is that the effectiveness of reasoning is maximized by minimizing the number of independence assumptions. If reasoning on N nodes, the way to minimize independence assumptions is to use the full inclusion-exclusion formula to calculate interdependencies between the N nodes. This involves 2N terms, one for each subset of the N nodes. Very rarely, in practical cases, will one have significant information about all these subsets. However, the nature of focused attention is that the system seeks to find out about as many of these subsets as possible, so as to be able to make the most accurate possible inferences, hence minimizing the use of unjustified independence assumptions. This implies that focused attention cannot hold too many items within it at one time, because if N is too big, then doing a decent sampling of the subsets of the N items is no longer realistic.

So, suppose that N items have been held within focused attention, meaning that a lot of predicates embodying combinations of N items have been constructed and evaluated and reasoned on. Then, during this extensive process of attentional focus, many of the N items will be useful in combination with each other — because of the existence of predicates joining the items. Hence, many HebbianLinks will grow between the N items — causing the set of N items to form a map.

By this reasoning, it seems that focused attention will implicitly be a map formation process — even though its immediate purpose is not map formation, but rather accurate inference (inference that minimizes independence assumptions by computing as many cross terms as is possible based on available direct and indirect evidence). Furthermore, it will encourage the formation of maps with a small number of elements in them (say, N<10). However, these elements may themselves be ConceptNodes grouping other nodes together, perhaps grouping together nodes that are involved in maps. In this way, one may see the formation of hierarchical maps, formed of clusters of clusters of clusters..., where each cluster has N<10 elements in it. These hierarchical maps manifest the abstract *dual network* concept that occurs frequently in OCP philosophy.

It is tempting to postulate that any intelligent system must display similar properties — so that focused attention, in general, has a strictly limited scope and causes the formation of maps that have *central cores* of roughly the same size as its scope. If this is indeed a general principle, it is an important one, because it tells you something about the general structure of derived hypergraphs associated with intelligent systems, based on the computational resource constraints of the systems.

The scope of an intelligent system's attentional focus would seem to generally increase logarithmically with the system's computational power. This follows immediately if one assumes that attentional focus involves free intercombination of the items within it. If attentional focus is the major locus of map formation, then — lapsing into SMEPH-speak — it follows that the bulk of the ConceptVertices in the intelligent system's derived hypergraphs may correspond to maps focused on a fairly small number of other ConceptVertices. In other words, derived hypergraphs may tend to have a fairly localized structure, in which each ConceptVertex has very strong InheritanceLinks pointing from a handful of other ConceptVertices (corresponding to the other things that were in the attentional focus when that ConceptVertex was formed).

Recognizing And Creating Self-Referential Structures

This is an overly brief page that covers a large and essential topic: how OCP system will be able to recognize and create large-scale self-referential structures in itself.

More raw material exists that will in future be used to expand this page into a more thorough treatment.

Essential Self-Referential Structures

Some of the most essential structures underlying human-level intelligence are self-referential in nature. These include

- the phenomenal self (see Thomas Metzinger's book "Being No One")
- the will
- reflective awareness

A casual but mathematical discussion of will and reflective awareness in terms of self-referential structures may be found at

http://www.goertzel.org/blog/2008/02/characterizing-consciousness-and-will.html

There, the following recursive definitions are given:

"S is conscious of X" is defined as: The declarative content that {"S is conscious of X" correlates with "X is a pattern in S"}

"S wills X" is defined as: The declarative content that {"S wills X" causally implies "S does X"}

Similarly, we may posit

"X is part of S's self" is defined as: The declarative content that {"X is a part of S's self" correlates with "X is a persistent pattern in S over time"}

(which has the fun, elegant implication that "self is to long-term memory as awareness is to short-term memory").

None of these are things that should be programmed into an artificial mind. Rather, they must emerge in the course of a mind's self-organization in connection with its environment.

However, a mind may be constructed so that, by design, these sorts of important self-referential structures are encouraged to emerge.

Encouraging the Recognition of Self-Referential Structures in the AtomTable

How can we do this — encourage an OCP instance to recognize complex self-referential structures that may exist in its AtomTable? This is important, because, according to the same logic as map formation: if these structures are explicitly recognized when they exist, they can then be reasoned on and otherwise further refined, which will then cause them to exist more definitively ... and hence to be explicitly recognized as yet more prominent patterns ... etc. The same virtuous cycle via which ongoing map recognition and encapsulation is supposed to lead to concept formation, may be posited on the level of complex self-referential structures, leading to their refinement, development and ongoing complexification.

One really simple way is to encode self-referential operators in the Combo vocabulary, that is used to represent the procedures grounding GroundedPredicateNodes.

That way, one can recognize self-referential patterns in the AtomTable via standard OCP methods like MOSES and IntegrativeProcedureAndPredicateLearning, so long as one uses Combo trees that are allowed to include self-referential operators at their nodes. All that matters is that one is able to take one of these Combo trees, compare it to an AtomTable, and assess the degree to which that Combo tree constitutes a pattern in that AtomTable.

But how can one do this? How can one match a self-referential structure like

```
EquivalenceLink
  EvaluationLink will (S,X)
  CausalImplicationLink
    EvaluationLink will (S,X)
    EvaluationLink do (S,X)
```

against an AtomTable or portion thereof?

note the shorthand notation; e.g.

```
EvaluationLink will (S,X)
```

really means

```
EvaluationLink
  will
  ListLink
    S
    X
```

Well, the question is whether there is some "map" of Atoms (some set of PredicateNodes) willMap, so that we may infer the SMEPH relationship

```
EquivalenceEdge
  EvaluationEdge willMap (S,X)
  CausalImplicationEdge
    EvaluationEdge willMap (S,X)
    EvaluationEdge doMap (S,X)
```

as a statistical pattern in the AtomTable's history over the recent past. (Here, doMap is defined to be the map corresponding to the built-in "do" predicate.)

If so, then this map willMap, may be encapsulated in a single new Node (call it willNode), which represents the system's will. This willNode may then be explicitly reasoned upon, used within concept creation, etc. It will lead to the spontaneous formation of a more sophisticated, fully-fleshed-out will map. And so forth.

Now, what is required for this sort of statistical pattern to be recognizable in the AtomTable's history? What is required is that EquivalenceEdges (which, note, must be part of the Combo vocabulary in order for any MOSES-related algorithms to recognize patterns involving them) must be defined according to the logic of hypersets rather than the logic of sets. What is fascinating is that this is no big deal! In fact, the AtomTable software structures support this automatically; it's just not the way most people are used to thinking about

things. In fact there is no reason, in terms of the AtomTable, not to create self-referential structures like the one given above.

The next question is how to we calculate the truth values of structures like the above, though. The truth value of a hyperset structure turns out to be an **infinite order probability distribution**, which is a funny and complex sort of mathematical beast, which however turns out not to be all that nasty in practice as one might expect. These probabilities are described at http://goertzel.org/papers/InfiniteOrderProbabilities.pdf . Infinite-order probability distributions are partially-ordered, and so one can compare the extent to which two different self-referential structures apply to a given body of data (e.g. an AtomTable), via comparing the infinite-order distros that constitute their truth values. In this way, one can recognize self-referential patterns in an AtomTable, and carry out encapsulaton of self-referential maps. This sounds very abstract and complicated, but the class of infinite-order distributions defined in the above-referenced papers actually have their truth values defined by simple matrix mathematics, so there is really nothing that abstruse involved in practice.

Clearly, with this highly speculative component of the OCP design we have veered rather far from anything the human brain could plausibly be doing in detail. For some ideas about how the brain might do this kind of stuff, take a look at the paper "How Might Probabilistic Logic Emerge from the Brain?" in the Proceedings of the AGI-08 conference. It is explored there how the brain may embody self-referential structures like the ones considered here, via using the hippocampus to encode whole neural nets as inputs to other neural nets. Regarding infinite-order probabilities, it is certainly the case that the brain is wired to carry out matrix manipulations, so that it's not completely outlandish to posit the brain could be doing something mathematically analogous. Thus, all in all, it does seem plausible to me that the brain could be doing something roughly analogous to what I've described here, even though the details would obviously be very different.

The ideas on this page have been fleshed out in more detail in various notes existing on Ben Goertzel's hard drive, and will be disbursed and polished in more detail when the time is right! At the moment (July 2008) there is a lot of more basic work on OCP to be done before we get to this stuff.

Chapter Seventeen: Toward a Theoretical Justification

Toward a Theoretical Justification of the OpenCogPrime Design

This page, and these three which it supervenes over

- OpenCogPrime:EssentialSynergies
- OpenCogPrime:PropositionsAboutMOSES
- OpenCogPrime:PropositionsAboutOpenCogPrime

address a conceptual and theoretical question: Why should anyone believe the OpenCogPrime design should actually work, in the strong sense of leading to general intelligence at the human level or ultimately beyond?

The question is addressed here from two different, closely related perspectives.

OpenCogPrime:EssentialSynergies pursues a qualitative argument regarding the power of the different OpenCogPrime:KnowledgeCreation algorithms in OCP to dramatically palliate each others' internal combinatorial explosions.

OpenCogPrime:PropositionsAboutMOSES and OpenCogPrime:PropositionsAboutOpenCogPrime take a somewhat different angle (building on the same conceptual basis), and pursue the question of how a mathematical study of the viability of the OpenCogPrime approach might be made. The mathematical study is left unfinished, and a number of interesting mathematical and quasi-mathematical propositions are formulated but not proved, so this aspect remains a tantalizing speculation. However, I (Ben Goertzel) feel it sheds conceptual light on aspects of the design, even without the (very significant) additional corroboration of mathematical proof.

We begin in OpenCogPrime:PropositionsAboutMOSES, not with OpenCogPrime in general, but by making some theoretical propositions regarding MOSES, the form of Probabilistic Evolutionary Learning used in the current OpenCogPrime system. The MOSES propositions are easier to grasp onto than the OpenCogPrime ones, and represent the same style of theorizing. Following these we proceed to OpenCogPrime:PropositionsAboutOpenCogPrime, which contains some more general propositions pertinent to the overall OpenCogPrime design.

A Note on Mathematical Theory

We now make a few comments on the role of mathematical formulation and proof in AGI.

The task of AGI design admits not only a multitude of technical approaches, but also a multitude of methodological approaches. Even within the scope of "computer science" approaches (as opposed to, e.g., "human brain emulation" approaches) a large degree of methodological variation is possible. For instance there is the distinction between "theory-based" and "pragmatics-based" approaches.

In a theory-based approach, one would begin with a mathematical theory of intelligence, and use this to arrive at an AGI design, and then use the mathematics to justify (either via rigorous proofs or hand-waving-laded physicist-style mathematical demonstrations) the design, and ideally to estimate its degree of intelligence given various amounts of computational resources.

In a pragmatics-based approach, on the other hand, one begins with an AGI design, convinces oneself that it looks plausible, and then builds it and experiments with it. Theories of its behavior may be constructed but the emphasis is on practice, and theory is generally used to resolve specific micro-level points regarding the behavior of the practical system.

The path to OpenCogPrime (via Novamente Cognition Engine, via the Webmind AI Engine, via various toy predecessor systems) began with an attempt to take the theory-based approach, but when this approach proved extremely slow and difficult, a shift to a more pragmatic approach was made. The OpenCogPrime design — like the NCE design to which it bears significant resemblance — was created via a synthesis of ideas from computer science, cognitive science, neuroscience, philosophy of mind and other areas, and the argument for its plausibility and practical viability is a complex one including some appeals to intuition along with more rigorous argumentation.

However, the more rigorous theoretical justification of designs in the OCP/NCE family is still a topic of interest, and does not seem an utter impossibility. We don't consider this kind of justification as necessary for the practical success of OpenCogPrime as an AGI system, but, we do consider it as potentially valuable in terms

of fine-tuning the design, understanding which parts are essential and which are less so, and identifying possible shortcomings of the design.

Essential Synergies Underlying OpenCogPrime Knowledge Creation

In the page OpenCogPrime:WhyItWillWork, an answer to "Why We Think the OCP Design Will Work" was given, and broken down into the following categories:

1. knowledge representation
2. cognitive architecture
3. teaching methodology
4. mirrorhouse of learning algorithms
5. capability for emergence of reflective self and attention

The propositions presented in the technical pages linked to from OpenCogPrime:TheoreticalJustification mainly address item 4 in this list: the mirrorhouse of learning algorithms. We believe all 5 of the above items need to be gotten right in order for an AGI to work, but, item 4 is probably the deepest and trickiest from a purely formal perspective. In essence, the OCP design has been created so that **if** the "mirrorhouse effect" works, **then** the combination of items 1-4 should emergently give rise to (item 5) reflective self and attention. On the other hand, if the mirrorhouse effect does not work, then it's unlikely that items 1-3 (which provide the "set-up", in a sense) are going to be able to lead to item 5.

Put informally, what the mirrorhouse effect means is that the different learning algorithms should help each other rather than impair each other. The issue may be put more technically as follows. Each of the key learning mechanisms underlying OCP is susceptible to combinatorial explosions. As the problems they confront become larger and larger, the performance gets worse and worse at an exponential rate, because the number of combinations of items that must be considered to solve the problems grows exponentially with the problem size. This could be viewed as a deficiency of the fundamental design, but we don't view it that way. Our view is that combinatorial explosion is intrinsic to intelligence. The task at hand is to dampen it sufficiently that realistically large problems can be solved, rather than to eliminate it entirely. One possible way to dampen it would be to design a single, really clever learning algorithm — one that was still susceptible to an exponential increase in computational requirements as problem size increases, but with a surprisingly small exponent. Another approach is the mirrorhouse approach: Design a bunch of learning algorithms, each focusing on different aspects of the learning process, and design them so that they each help to dampen each others' combinatorial explosions. This is the approach taken within OCP. The component algorithms are clever on their own — they are less susceptible to combinatorial explosion than many competing approaches in the narrow-AI literature. But the real meat of the design lies in the intended interactions between the components.

To see what this means more specifically, let's review some of the key components of OCP as discussed.

Synergies that Help Inference

Probabilistic Logic Networks, for starters. The combinatorial explosion in PLN is obvious: forward and backward chaining inference are both fundamentally explosive processes, reined in only by pruning heuristics. This means that for nontrivially complex inferences to occur, one needs **really, really clever** pruning heuristics. The OCP design combines simple heuristics with pattern mining, MOSES and economic attention allocation as pruning heuristics. Economic attention allocation assigns importance levels to Atoms, which helps guide

pruning. Greedy pattern mining is used to search for patterns in the stored corpus of inference trees, to see if there are any that can be used as analogies for the current inference. And MOSES comes in when there is not enough information (from importance levels or prior inference history) to make a choice, yet exploring a wide variety of available options is unrealistic. In this case, MOSES tasks may be launched, pertinently to the leaves at the fringe of the inference tree, under consideration for expansion. For instance, suppose there is an Atom _A_ at the fringe of the inference tree, and its importance hasn't been assessed with high confidence, but a number of items B are known so that

```
MemberLink A B
```

Then, MOSES maybe used to learn various relationships characterizing A, based on recognizing patterns across the set of B that are suspected to be members of A. These relationships may then be used to assess the importance of A more confidently, or perhaps to enable the inference tree to match one of the patterns identified by pattern mining on the inference tree corpus. For example, if MOSES figures out that

```
SimilarityLink G A
```

then it may happen that substituting G in place of A in the inference tree, results in something that pattern mining can identify as being a good (or poor) direction for inference.

Synergies that Help MOSES

MOSES's combinatorial explosion is obvious: the number of possible programs of size N increases very rapidly with N. The only way to get around this is to utilize prior knowledge, and as much as possible of it. When solving a particular problem, the search for new solutions must make use of prior candidate solutions evaluated for that problem, and also prior candidate solutions (including successful and unsuccessful ones) evaluated for other related problems.

But, extrapolation of this kind is in essence a contextual analogical inference problem. In some cases it can be solved via fairly straightforward pattern mining; but in subtler cases it will require inference of the type provided by PLN. Also, attention allocation plays a role in figuring out, for a given problem A, which problems B are likely to have the property that candidate solutions for B are useful information when looking for better solutions for A.

Synergies that Help Attention Allocation

Economic attention allocation, without help from other cognitive processes, is just a very simple process analogous to "activation spreading" and "Hebbian learning" in a neural network. The other cognitive processes are the things that allow it to more sensitively understand the attentional relationships between different knowledge items (e.g. which sorts of items are often usefully thought about in the same context, and in which order).

Further Synergies Related to Pattern Mining

Statistical, greedy pattern mining is a simple process, but it nevertheless can be biased in various ways by other, subtler processes.

For instance, if one has learned a population of programs via MOSES, addressing some particular fitness function, then one can study which items tend to be utilized in the same programs in this population. One may then direct pattern mining to find patterns combining these items found to be combined in the MOSES population. And conversely, relationships denoted by pattern mining may be used to probabilistically bias the models used within MOSES.

Statistical pattern mining may also help PLN by supplying it with information to work on. For instance, conjunctive pattern mining finds conjunctions of items, which may then be combined with each other using PLN, leading to the formation of more complex predicates. These conjunctions may also be fed to MOSES as part of an initial population for solving a relevant problem.

Finally, the main interaction between pattern mining and MOSES/PLN is that the former may recognize patterns in links created by the latter. These patterns may then be fed back into MOSES and PLN as data. This virtuous cycle allows pattern mining and the other, more expensive cognitive processes to guide each other. Attention allocation also gets into the game, by guiding statistical pattern mining and telling it which terms (and which combinations) to spend more time on.

Synergies Related to Map Formation

The essential synergy regarding map formation is obvious: Maps are formed based on the HebbianLinks created via PLN and simpler attentional dynamics, which are based on which Atoms are usefully used together, which is based on the dynamics of the cognitive processes doing the "using." On the other hand, once maps are formed and encapsulated, they feed into these other cognitive processes. This synergy in particular is critical to the emergence of self and attention.

What has to happen, for map formation to work well, is that the cognitive processes must *utilize* encapsulated maps in a way that gives rise overall to relatively clear clusters in the network of HebbianLinks. This will happen if the encapsulated maps are not too complex for the system's other learning operations to understand. So, there must be useful coordinated attentional patterns whose corresponding encapsulated-map Atoms are not too complicated. This has to do with the system's overall parameter settings, but largely with the settings of the attention allocation component. For instance, this is closely tied in with the limited size of "attentional focus" (the famous 7 +/- 2 number associated with humans' and other mammals short term memory capacity). If only a small number of Atoms are typically very important at a given point in time, then the maps formed by grouping together all simultaneously highly important things will be relatively small predicates, which will be easily reasoned about — thus keeping the "virtuous cycle" of map formation and comprehension going effectively.

Conclusion

The above synergies, plainly, are key to the proposed functionality of the OCP system. Without them, the cognitive mechanisms are not going to work adequately well, but are rather going to succumb to combinatorial explosions.

The other aspects of OCP — the cognitive architecture, the knowledge representation, the embodiment framework and associated developmental teaching methodology — are all critical as well, but none of these will yield the critical emergences of intelligence without cognitive mechanisms that effectively scale. And, in the

absence of cognitive mechanisms that effectively scale *on their own*, we must rely on cognitive mechanisms that *effectively help each other to scale.*

The reasons why we believe these synergies will exist are essentially qualitative: we have not proved theorems regarded these synergies, and we have observed them in practice only in simple cases so far. However, we do have some ideas regarding how to potentially prove theorems related to these synergies, and some of these are described in other wiki pages linked to from Theoretical Justification.

Propositions about MOSES

Why is MOSES a good approach to automated program learning? The conceptual argument in favor of MOSES may be broken down into a series of propositions, which are given here both in informal "slogan" form and in semi-formalized "proposition" form.

Note that the arguments given here appear essentially applicable to other MOSES-related algorithms such as Pleasure as well. The page however was originally written in regard to MOSES and hasn't been revised in the light of the creation of Pleasure.

Slogan 1 refers to "ENF", Elegant Normal Form, which is used by MOSES as a standard format for program trees. This is a way that MOSES differs from GP for example: GP does not typically normalize program trees into a standard syntactic format, but leaves trees heterogeneous as to format.

Proposition: ENF Helps to Guide Syntax-Based Program Space Search

Slogan 1

Iterative optimization is guided based on syntactic distance ==> ENF is good

Proposition 1

On average, over a class C of fitness functions, it is better to do optimization based on a representation in which the (average over all functions in C of the) correlation between syntactic and semantic distance is larger. This should hold for any optimization algorithm which makes a series of guesses, in which the new guesses are chosen from the old ones in a way that is biased to choose new guesses that have small syntactic distance to the old one.

Note that GA, GP, BOA, BOAP and MOSES all fall into the specified category of optimization algorithms

It is not clear what average smoothness condition is useful here. For instance, one could look at the average of $d(f(x),f(y))/d(x,y)$ for $d(x,y)<A$, where d is syntactic distance and A is chosen so that the optimization algorithm is biased to choose new guesses that have syntactic distance less than A from the old ones.

Proposition 2: Demes are Useful if Syntax/Semantics Correlations in Program Space Have a Small Scale

This proposition refers to the strategy of using "demes" in MOSES: instead of just evolving one population of program trees, a collection of "demes" are evolved, each one a population of program trees that are all somewhat similar to each other.

Slogan 2 Small-scale syntactic/semantic correlation ==> demes are good [If the maximal syntactic/semantic correlation occurs on a small scale, then multiple demes are useful]

Proposition 2 Let d denote syntactic distance, and d1 denote semantic distance. Suppose that the correlation between d(x,y) and d1(x,y) is much larger for d(x,y)<A than for A<d(x,y)<2A or A<d(x,y), as an average across all fitness functions in class C. Suppose the number of spheres of radius R required to cover the space of all genotypes is n(R). Then using n(R) demes will provide significantly faster optimization than using n(2R) demes or 1 deme. Assume here the same conditions on the optimization algorithm as in Proposition 1.

Proposition 2.1 Consider the class of fitness functions defined by

```
Correlation( d(x,y), d1(x,y) || d(x,y) = a ) = b
```

Then, across this class, there is a certain number D of demes that will be optimal on average.... I.e. the optimal number of demes depends on the scale-dependence of the correlation between syntactic & semantic distance....

Proposition 3: Probabilistic Program Tree Modeling Helps in the Presence of Cross-Modular Dependencies

This proposition refers to the use of BOA-type program tree modeling within MOSES. What it states is that this sort of modeling is useful if the programs in question have significant cross-modular dependences that are not extremely difficult to detect.

Slogan 3 Cross-modular dependencies ==> BOA is good [If the genotypes possess significant internal dependencies that are **not** concordant with the genotypes' internal modular structure, then BOA-type optimization will significantly outperform GA/GP-type optimization for deme-exemplar extension.]

Proposition 3 Consider the classification problem of distinguishing fit genotypes from less fit genotypes, within a deme. If significantly greater classification accuracy can be obtained by classification rules containing "cross-terms" combining genotype elements that are distant from each other within the genotypes — and these cross-terms are not too large relative to the increase in accuracy they provide — then BOA-type modeling will significantly outperform GA/GP-type optimization.

The catch in Proposition 3 is that the BOA-type modeling must be sophisticated enough to recognize the specific cross-terms involved, of course.

Proposition 4: Relating ENF to BOA

Now, how does BOA learning relate to ENF?

Proposition 4 ENF decreases, on average, the number and size of cross-terms in the classification rules mentioned in Proposition 3.

Conclusion Regarding Speculative MOSES Theory

What we see from the above is that:

- ENF is needed in order to make the fitness landscape smoother, but can almost never work perfectly so there will nearly always be some long-distance dependencies left after ENF-ization
- The smoother fitness landscape enabled by ENF, enables optimization using demes and incremental exemplar-expansion to work, assuming the number of demes is chosen intelligently
- Within a deme, optimization via incremental exemplar growth is more efficient using BOA than straight evolutionary methods, due to the ability of BOA to exploit the long-distance dependencies not removed by ENF-ization

These propositions appear to capture the basic conceptual justification for the current MOSES methodology. Of course, proving them will be another story, and will likely involve making the proposition statements significantly more technical and complex.

Another interesting angle on these propositions is to view them as constraints on the *problem type* to which MOSES may be fruitfully applied. Obviously, no program learning algorithm can outperform random search on random program learning problems. MOSES, like any other algorithm, needs to be applied to problems that match its particular biases. What sorts of problems match MOSES's biases?

In particular, the right question to ask is: Given a particular choice regarding syntactic program representation, what sorts of problems match MOSES's biases as induced by this choice?

If the above propositions are correct, the answer is, basically: Problems for which semantic distance (distance in fitness) is moderately well-correlated with syntactic distance (in the chosen representation) over a short scale but not necessarily over a long scale, and for which a significant percentage of successful programs have a moderate but not huge degree of internal complexity (as measured by internal cross-module dependencies).

Implicit in this is an explanation of why MOSES, on its own, is likely not a good approach to solving extremely large and complex problems. This is because for an extremely large and complex problem, the degree of internal complexity of successful programs will likely be too high for BOA modeling to cope with. So then, in these cases MOSES will effectively operate as a multi-start local search on normalized program trees, which is not a stupid thing, but unlikely to be adequately effective for most large, complex problems.

We see from the above that even in the case of MOSES, which is much simpler than OCP, formulating the appropriate theory adequately is not a simple thing, and proving the relevant propositions may be fairly difficult. However, we can also see from the MOSES example that the creation of a theoretical treatment does have some potential for clarifying the nature of the algorithm and its likely range of applicability.

Propositions About OCP

On this page we present some speculations regarding the extension of the approach to MOSES-theory presented in OpenCogPrime:PropositionsAboutMOSES to handle OCP in general. This is of course a much more

complex and subtle matter, yet we suggest that in large part it may be handled in a similar way. This way of thinking provides a different perspective on the OCP design — one that has not yet substantially impacted the practical aspects of the design, but may well be of use to us as we iteratively refine the design in the future, in the course of testing and teaching OCP AGI systems.

As with the propositions in previous section but even more so, the details of these heuristic propositions will likely change a fair bit when/if rigorous proof/statement is attempted. But we are intuitively fairly confident that the basic ideas described here will hold up to rigorous analysis.

Finally, one more caveat: the set of propositions listed here is not presented as *complete*. By no means! A complete theoretical treatment of OCP, along these lines, would involve a more substantial list of related propositions. The propositions given here are meant to cover a number of the most key points, and to serve as illustrations of the sort of AGI theory we believe/hope may be possible to do in the near and medium term future.

When PLN Inference Beats BOA

This proposition explains why, in some cases, it will be better to use PLN rather than BOA within MOSES, for modeling the dependencies within populations of program trees.

Slogan 5 Complex cross-modular dependencies which have similar nature for similar fitness functions ==> PLN inference is better than BOA for controlling exemplar extension

Proposition 5 Consider the classification problem of distinguishing fit genotypes from less fit genotypes, within a deme. If

- significantly greater classification accuracy can be obtained by classification rules containing "cross-terms" combining genotype elements that are distant from each other within the genotypes, but
- the search space for finding these classification rules is tricky enough that a greedy learning algorithm like decision-tree learning (which is used within BOA) isn't going to find the good ones
- the classification rules tend to be similar, for learning problems for which the fitness functions are similar

Then, PLN will significantly outperform BOA for exemplar extension within MOSES, due to its ability to take history into account.

Conditions for the Usefulness of Hebbian Inference Control

Now we turn from MOSES to PLN proper. The approximate probabilistic correctness of PLN is handled via PLN theory itself, as presented in the PLN book. However, the trickiest part of PLN in practice is *inference control*, which in the OCP design is proposed to be handled via "experiential learning." This proposition pertains to the conditions under which Hebbian-style, inductive PLN inference control can be useful.

Slogan 6 If similar theorems generally have similar proofs, then inductively-controlled PLN can work effectively

Proposition 6

- Let L_0 = a simple "base level" theorem-proving framework, with fixed control heuristics
- For n>0, let L_n = theorem-proving done using L_(n-1), with inference control done using data mining over a DB of inference trees, utilizing L_(n-1) to find recurring patterns among these inference trees that are potentially useful for controlling inference

Then, if T is a set of theorems so that, within T, theorems that are similar according to "similarity provable in L_(n-1) using effort E" have proofs that are similar according to the same measure, then L_n will be effective for proving theorems within T

Clustering-together of Smooth Theorems

This proposition is utilized within Theorem 8, below, which again has to do with PLN inference control.

Slogan 7 "Smooth" theorems tend to cluster together in theorem-space

Proposition 7 Define the smoothness of a theorem as the degree to which its proof is similar to the proofs of other theorems similar to it. Then, smoothness varies smoothly in theorem-space. I.e., a smooth theorem tends to be close-by to other smooth theorems.

When PLN is Useful Within MOSES

Above it was argued that PLN is useful within MOSES due to its capability to take account of history (across multiple fitness functions). But this is not the only reason to utilize PLN within MOSES; Propositions 6 and 7 above give us another theoretical reason.

Proposition 8 If similar theorems of the form "Program A is likely to have similar behavior to program B" tend to have similar proofs, and the conditions of Slogan 6 hold for the class of programs in question, then inductively controlled PLN is good (and better than BOA) for exemplar extension. (This is basically Proposition 6 + Proposition 7)

When MOSES is Useful Within PLN

We have explored theoretical reasons why PLN should be useful within MOSES, as a replacement for the BOA step used in the standalone implementation of MOSES. The next few propositions work in the opposite direction, and explore rasons why MOSES should be useful within PLN, for the specific problem of finding elements of a set given a qualitative (intensional) description of a set. (This is not the only use of MOSES for helping PLN, but it is a key use and a fairly simple one to address from a theoretical perspective.)

Proposition 9 In a universe of sets where intensional similarity and extensional similarity are well-correlated, the problem of finding classification rules corresponding to a set S leads to a population of decently fit candidate solutions with high syntactic/semantic correlation [so that demes are good for this problem]

Proposition 10: In a universe of sets satisfying Proposition 9, where sets have properties with complex interdependencies, BOA will be useful for exemplar extension (in the context of using demes to find classification rules corresponding to sets)

Proposition 11: In a universe of sets satisfying Proposition 10, where the interdependencies associated with a set S's property-set vary "smoothly" as S varies, working inference is better than BOA for exemplar extension

Proposition 12: In a universe of sets satisfying Proposition 10, where the proof of theorems of the form "Both the interdependencies of S's properties, and the interdependencies of T's properties, satisfy predicate F" depends smoothly on the theorem statement, then inductively controlled PLN will be effective for exemplar extension

On the Smoothness of Some Relevant Theorems

We have talked a bit about smooth theorems, but what sorts of theorems will tend to be smooth? If the OCP design is to work effectively, the "relevant" theorems must be smooth; and the following proposition gives some evidence as to why this may be the case.

Proposition 13 In a universe of sets where intensional similarity and extensional similarity are well-correlated, probabilistic theorems of the form "A is a probabilistic subset of B" and "A is a pattern in B" tend to be smooth....

Note that: For a set S of programs, to say "intensional similarity and extensional similarity are well-correlated" among subsets of S, means the same thing as saying that syntactic and semantic similarity are well-correlated among members of S

Proposition 14 The set of motor control programs, for a set of standard actuators like wheels, arms and legs, displays a reasonable level of correlation between syntactic and semantic similarity

Proposition 15 The set of sentences that are legal in English displays a high level of correlation between syntactic and semantic similarity.

(The above is what, in Chaotic Logic, I called the "principle of continuous compositionality", extending Frege's Principle of Compositionality. It implies that language is learnable via OCP-type methods.... Unlike the other Propositions formulated here, it is more likely to be addressable via statistical than formal mathematical means; but insofar as English syntax can be formulated formally, it may be considered a *roughly-stated) mathematical proposition.)

Recursive Use of "MOSES+PLN" to Help With Attention Allocation

Proposition 16 The set of propositions of the form "When thinking about A is useful, thinking about B is often also useful" tends to be smooth — if "thinking" consists of MOSES plus inductively controlled PLN, and the universe of sets is such that this cognitive approach is generally a good one

This (Prop. 16) implies that adaptive attention allocation can be useful for a MOSES+PLN system, if the attention allocation itself utilizes MOSES+PLN

The Value of Conceptual Blending

Proposition 17: In a universe of sets where intensional similarity and extensional similarity are well-correlated, if two sets A and B are often useful in proving theorems of the form "C is a (probabilistic) subset of D", then "blends" of A and B will often be useful for proving such theorems as well.

This is a justification of conceptual blending for concept formation

A Justification of Map Formation

Proposition 18: If a collection of terms A is often used together in MOSES+PLN, then similar collections B will often be useful as well, for this same process ... assuming the universe of sets is so that intensional and extensional similarity are correlated, and MOSES+PLN works well

This is a partial justification of map formation, in that finding collections B similar to A is achieved by encapsulating A into a node A' and then doing reasoning on A'

Concluding Remarks

The above set of propositions is certainly not complete. For instance, one might like to throw in conjunctive pattern mining as a rapid approximation to MOSES; and some specific justification of artificial economics as a path to effectively utilizing MOSES/PLN for attention allocation; etc.

But, overall, it seems fair to say that the above set of propositions smells like a possibly viable path to a theoretical justification of the OCP design.

To summarize the above ideas in a nutshell, we may say that the effectiveness of the OCP design appears intuitively to follow from the assumptions that:

- within the space of relevant learning problems, problems defined by similar predicates tend to have somewhat similar solutions
- according to OCP's knowledge representation, procedures and predicates with very similar behaviors often have very similar internal structures, and vice versa (and this holds to a drastically lesser degree if the "very" is removed)
- for relevant theorems ("theorems" meaning Atoms whose truth values need to be evaluated, or whose variables or SatisfyingSets need to be filled in, via PLN): similar theorems tend to have similar proofs, and the degree to which this holds varies smoothly in proof-space
- the world can be well modeled using sets for which intensional and extensional similarity are well correlated: meaning that the mind can come up with a system of "extensional categories" useful for describing the world, and displaying characteristic patterns that are not too complex to be recognized by the mind's cognitive methods

To really make use of this sort of theory, of course, two things would need to be done. For one thing, the propositions would have to be proved (which will probably involve some serious adjustments to the proposition statements). For another thing, some detailed argumentation would have to be done regarding why the "relevant problems" confronting an embodied AGI system actually fulfill the assumptions. This might turn out to be the hard part, because the class of "relevant problems" is not so precisely defined. For very specific problems like — to name some examples quasi-randomly — natural language learning, object recogntion, learning to navigate

in a room with obstacles, or theorem-proving within a certain defined scope, however, it may be possible to make detailed arguments as to why the assumptions should be fulfilled.

Recall that what makes OCP different from huge-resources AI designs like AIXI (including AIXI-tl) and the GodelMachine is that it involves a number of specialized components, each with their own domains and biases and some with truly general potential as well, hooked together in an integrative architecture designed to foster cross-component interaction and overall synergy and emergence. The strength and weakness of this kind of architecture is that it is specialized to a particular class of environments. AIXItl and the Godel Machine can handle any type of environment roughly equally well (which is: very, very slowly), whereas, OCP has the potential to be much faster when it's in environment that poses it learning problems that match its particular specializations. What we have done in the above series of propositions is to partially formalize the properties an environment must have to be "OCP-friendly." If the propositions are essentially correct, and if interesting real-world environments largely satisfy their assumptions, then OCP is a viable AGI design.