

Chapter 2

Knowledge Representation Using the AtomSpace

2.1 Introduction

CogPrime’s knowledge representation must be considered on two levels: implicit and explicit. This chapter considers mainly explicit knowledge representation, with a focus on representation of declarative knowledge. We will describe the Atom knowledge representation, a generalized hypergraph formalism which comprises a specific vocabulary of Node and Link types, used to represent declarative knowledge but also, to a lesser extent, other types of knowledge as well. Other mechanisms of representing procedural, episodic, attentional, and intentional knowledge will be handled in later chapters, as will the subtleties of implicit knowledge representation.

The AtomSpace Node and Link formalism is the most obviously distinctive aspect of the OpenCog architecture, from the point of view of a software developer building AI processes in the OpenCog framework. But yet, the features of CogPrime that are most important, in terms of our theoretical reasons for estimating it likely to succeed as an advanced AGI system, are not really dependent on the particulars of the AtomSpace representation.

What’s important about the AtomSpace knowledge representation is mainly that it provides a flexible means for compactly representing multiple forms of knowledge, in a way that allows them to interoperate—where by “interoperate” we mean that e.g. a fragment of a chunk of declarative knowledge can link to a fragment of a chunk of attentional or procedural knowledge; or a chunk of knowledge in one category can overlap with a chunk of knowledge in another category (as when the same link has both a (declarative) truth value and an (attentional) importance value). In short, any representational infrastructure sufficiently flexible to support

- compact representation of all the key categories of knowledge playing dominant roles in human memory
- the flexible creation of specialized sub-representations for various particular subtypes of knowledge in all these categories, enabling compact and rapidly manipulable expression of knowledge of these subtypes

- the overlap and interlinkage of knowledge of various types, including that represented using specialized sub-representations

will probably be acceptable for CogPrime’s purposes. However, precisely formulating these general requirements is tricky, and is significantly more difficult than simply articulating a single acceptable representational scheme, like the current OpenCog Atom formalism. The Atom formalism satisfies the relevant general requirements and has proved workable from a practical software perspective.

In terms of the Mind-World Correspondence Principle introduced in Chap. 10 of Part 1, the important point regarding the Atom representation is that it must be flexible enough to allow the compact and rapidly manipulable representation of knowledge that has aspects spanning the multiple common human knowledge categories, in a manner that allows easy implementation of cognitive processes that will manifest the Mind-World Correspondence Principle in everyday human-like situations. The actual manifestation of mind-world correspondence is the job of the cognitive processes acting on the AtomSpace—the job of the AtomSpace is to be an efficient and flexible enough representation that these cognitive processes can manifest mind-world correspondence in everyday human contexts given highly limited computational resources.

2.2 Denoting Atoms

First we describe the textual notation we’ll use to denote various sorts of Atoms throughout the following chapters. The discussion will also serve to give some particular examples of cognitively meaningful Atom constructs.

2.2.1 *Meta-Language*

As always occurs when discussing (even partially) logic-based systems, when discussing CogPrime 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 CogPrime are equivalent, and this is different from stating that CogPrime 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 CogPrime structures or dynamics here, there is no need to formally specify the logic being used for the 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 (IntensionalInheritance 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 CogPrime. The former refers to an equals relationship observed by the authors of the book, but perhaps never represented explicitly inside CogPrime.

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

```
InheritanceLink A B <sAB>
InheritanceLink B C <sBC>
| -
InheritanceLink 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
```

```
ExtensionalImplicationLink_HOJ
```

```
AND
```

```
InheritanceLink $A $B <$sAB>
```

```
InheritanceLink $B $C <$sBC>
```

```
AND
```

```
InheritanceLink $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 CogPrime. In the latter expression the variables \$A, \$sAB, and so forth represent actual Variable Atoms within CogPrime. In the former expression the variables represent concrete, non-Variable Atoms within CogPrime, which however are being considered as variables within the metalanguage.

(As explained in the PLN book, a link labeled with “HOJ” refers to a “higher order judgment”, meaning a relationship that interprets its relations as entities with particular truth values. For instance,

```
ImplicationLink_HOJ
```

```
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).

2.2.2 Denoting Atoms

Atoms are the basic objects making up CogPrime knowledge. They come in various types, and are associated with various dynamics, which are embodied in Mind Agents. 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.

2.2.2.1 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 CogPrime 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, this may happen for instance if the Atom is considered in a different context (via a ContextLink), each version has a VersionHandle, so that accessing an AtomVersion requires specifying an AtomHandle plus a VersionHandle. See Chap. 1 for more information on Handles.

OpenCog never assigns Atoms names *on its own*; in fact, Atom names are assigned only in the two sorts of cases 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 write about and 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 with Handle 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.

2.2.2.2 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.

Another option currently implemented in OpenCog is to explicitly restrict the type of a variable using `TypedVariableLink` such as follows

```
TypedVariableLink
  VariableNode $X
  VariableTypeNode "ConceptNode"
```

Note also that we will frequently remove the suffix `Link` or `Node` from their type name, such as

```
Inheritance
  Concept A
  Concept B
```

instead of

```
InheritanceLink
  ConceptNode A
  ConceptNode B
```

2.2.2.3 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* (see Chap. 16). 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 A in the context C, we can use the construct

```
ContextLink <truth value>
  C
  A
```

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.

We may also put the TruthValue indicator in a different place, e.g. using indent notation,

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

This is mostly useful when dealing with long and complicated constructions.

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.

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

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 and recalled in Chap. 16. 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).*

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

```
( Intensional Inheritance 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 here and the <> notation will serve us well.

2.2.2.4 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, but not only, 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>>`

or simply

`Cow_7 <<.1, .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

```
(Intensional InheritanceLink Ben monster).AttentionValue
      = [ (STI, .1), (LTI, .8) ]
```

where appropriate.

2.2.2.5 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. [WIKISOURCE:FunctionNotation](#)

2.3 Representing Functions and Predicates

`SchemaNodes` and `PredicateNodes` contain functions internally; and `Links` may also usefully be considered as functions. We now briefly discuss the representations and notations we will use to indicate 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).

2.3.0.6 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

$$\begin{array}{c} @ \\ / \quad \backslash \\ f \quad \quad x \end{array}$$

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)

2.3.1 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 link, writing e.g.

```
atTimeLink
  T
  ExecutionLink S A B
```

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 function.

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 that would not be necessary if the CogPrime system were revised so that all cognitive functions were carried out using SchemaNodes. However, in the current CogPrime version, where most cognitive functions are carried out using C++ MindAgent objects, if we want CogPrime 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 rules.

2.3.1.1 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 <tv>
```

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.¹

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
```

Note how the variables have been stripped out of the expression, see the PLN book for more explanation about that. 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
```

¹ Actually, if *P* does take some inputs, *EvaluationLink P <tv>* is defined too and *tv* corresponds to the average of *P(X)* over all inputs *X*, this is explained in more depth in the PLN book.

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.

2.3.2 Denoting Schema and Predicate Variables

CogPrime 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 CogPrime:

- 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 $\text{eats}(\text{cat}, X)$.

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 (which is not commutative but that is not the point), stored in the `ProcedureRepository` and linked to the `PredicateNode` for f . These f -internal variables, which are expressed here using the letters a and b , are not `VariableNodes` in the `CogPrime 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)
```

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.

2.3.2.1 Links as Predicates

It is conceptually important to recognize that `CogPrime` 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 $X animal
```

acting on the input

cat

or the predicate

InheritanceLink \$X \$Y

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 likely create a revision of CogPrime 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.

2.3.3 Variable and Combinator Notation

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

This section deals with the representation of *variable expressions* in a variable-free way, in a CogPrime 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.

2.3.3.1 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 CogPrime lingo) without any variables in them.

As will be discussed later (e.g. Chap. 15 on evolutionary learning, although the same process is also useful for supporting probabilistic reasoning on procedures), in 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 to 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).

In large part, we want to remove variables so we can use functional programming tools to normalize programs into a standard and pretty hierarchical form, in order to 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.

2.3.3.2 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 (from combinatory logic), 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.

WIKISOURCE:TypeInheritance

2.3.4 Inheritance Between Higher-Order Types

Next, this section 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.

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

$T1 \rightarrow S1 \text{ inh } T2 \rightarrow S2$

iff

$T2 \text{ inh } T1$

$S1 \text{ inh } S2$

In other words, we assume higher-order type inheritance is countervariant. The reason is that, if $R1 = T1 \rightarrow S1$ is to be a special case of $R2 = T2 \rightarrow S2$, 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 CogPrime, the greater logical consistency of the countervariance approach holds sway.

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

$(T1, S1) \text{ inh } (T2, S2)$

iff

$T1 \text{ inh } T2$

$S1 \text{ inh } S2$

As a mnemonic formula, we may say

(general \rightarrow specific) inherits from (specific \rightarrow 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

$[1 \text{ inh } 2]$

to denote that

$Xv1 \text{ inh } Xv2$

So for example,

$(\text{INT}, \text{VOID}) \text{ inh } (Xv1, Xv2)$

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

On the other hand,

$(\text{INT}, \text{VOID}) \text{ inh } (Xv1, Xv2), [1 \text{ inh } 2]$

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

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

- $\text{INT} \text{ inh } \text{NUMBER}$
- $\text{FLOAT} \text{ inh } \text{NUMBER}$
- $\text{CHAR} \text{ inh } \text{STRING}$
- $(\text{NO} \rightarrow \text{FL}) \text{ inh } (\text{INT} \rightarrow \text{FL})$
- $(\text{FL} \rightarrow \text{INT}) \text{ inh } (\text{FL} \rightarrow \text{NO})$
- $((\text{INT} \rightarrow \text{FL}) \rightarrow (\text{FL} \rightarrow \text{INT})) \text{ inh } ((\text{NO} \rightarrow \text{FL}) \rightarrow (\text{FL} \rightarrow \text{NO}))$.

2.3.5 Advanced Schema Manipulation

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

2.3.5.1 Listification

First, there are two ways to represent n-ary relations in CogPrime'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 $(X - 1)^2$. If we use the notations—and *pow* to denote SchemaNodes embodying the corresponding operations, 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* and *B* combinators a couple times to move the variable *X* to the front. The *B* combinator (see Combinatory Logic REF) is recalled below:

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

This is accomplished as follows (using the standard convention of left-associativity for the application operator, denoted @ in the tree representation given in Sect. Execution Output Links)

```
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 CogPrime would look like

```
ExOutLink
  ExOutLink
    B
    ExOutLink
      ExOutLink
        C
```

```

                ExOutLink
                  unlistify
                    pow
                2
ExOutLink
  ExOutLink
    C
    ExOutLink
      unlistify
        -
      1

```

The main thing to be observed 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
  ExOutLink
    B
    ExOutLink
      ExOutLink
        C
        pow_curried
      2
    ExOutLink
      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 13 lines whereas the variable-bearing version is 9 lines, a minor increase in length that brings a lot of operational simplification.

2.3.5.2 Argument Permutation

In dealing with List relationships, there will sometimes be use for an argument-permutation operator, let us call it *P*, defined as follows

$$(P \ p \ f) \ (v1, \dots, vn) = f \ (p \ (v1, \dots, vn))$$

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 (P {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.

Finally, let's conclude with a similar example in a more standard predicate logic notation, involving both combinators and the permutation argument operator introduced above. We will translate the variable-laden predicate

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

into the equivalent combinatory logic tree. Let us first recall the combinator `S` whose function is to distribute an argument over two terms.

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

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 (P {2,1} likes))) likes
```

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

```
S (B AND (B (P {2,1} likes))) likes (x,y)
```

`S` gets evaluated first, to produce

```
(B AND (B (P {2,1} likes)) (x,y)) (likes (x,y))
```

now the first `B`

```
AND ((B (P {2,1} likes)) (x,y)) (likes (x,y))
```

now the second one

```
AND ((P {2,1} likes) (x,y)) (likes (x,y))
```

now `P`

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

which is what we wanted.

Engineering General Intelligence, Part 2
The CogPrime Architecture for Integrative, Embodied
AGI

Goertzel, B.; Pennachin, C.; Geisweiller, N.
2014, XXII, 562 p. 42 illus., 9 illus. in color., Hardcover
ISBN: 978-94-6239-029-4
A product of Atlantis Press