CENTER FOR
Brains
Minds+
Machines

July 11th, 2020

A Definition of General Problem Solving

by

Qianli Liao and Tomaso Poggio

Center for Brains, Minds, and Machines, McGovern Institute for Brain Research,
Massachusetts Institute of Technology, Cambridge, MA, 02139.

Abstract: What is general intelligence? What does it mean by general problem solving? We attempt to give a definition of general problem solving, characterize the common process of problem solving and provide a basic algorithm that can in principle solve a wide range of novel tasks. Specifically, we represent general problem solving as a information/data conversion task that can be solved by finding dependencies/explanations. We propose "Object-Oriented Programming", a general reasoning framework with object-centric operations that solves problems in a human-like goal-driven fashion, guided by information, compositionality and general theories of objects, instead of merely via large scale searches.

[1]

This work was supported by the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF - 1231216.

[1] Some of the ideas were mentioned earlier in the appendix of [Liao and Poggio, 2020].

Contents

1 Introduction

In recent years, we have developed machines that can solve problems very well on many domains separately. They are often trained or engineered on each domain intensively to achieve this level of proficiency. However, they still do not exhibit the any "domain-general" intelligence: that is, when facing a novel problem, they often cannot solve it. They lack out-of-distribution generalization as described in [Liao and Poggio, 2020]. They do not have any "general problem solving" skills that humans apparently have.

In this report, we try to define what it means to solve a problem in general. We breakdown the process of solving a general problem into detailed, mathematically well-defined and machine executable steps. With this framework, theories and algorithms can be developed to solve general problems.

2 General Problem Solving as Information/Data Conversion

We can define general problem solving as an information or data conversion problem from input I to output O with memory M. For simplicity, we focus on training with only a few examples of input-output pairs $(I_1, O_1), (I_2, O_2), ..., (I_n, O_n)$.

The information conversion framework is a process of generating the output using some information present in the input. This process must be carried out by three components:

- Extract some information from the input
- Convert the information in some way
- Express the resultant information in some format.

As one can see, these three steps are mathematically better defined analogues of **reading**, **thinking** and **writing**.

Since almost all tasks we encounter can be solved by a combination of reading, thinking and writing, the information conversion framework should be expressive enough to be a general problem solving framework.

Note that all of these three processes may involve some memory M of the intelligent system.

This framework was also discussed in [Liao et al., 2020] and [Liao and Poggio, 2020].

In the following sections, we show detailed steps of general problem solving that we believe are adopted unconciously by humans. We describe these steps in well-defined language so that they should be machine implementable.

2.1 A List of Relevant Concepts

It is impossible to algorithmically address the vast problem of general problem solving without involving a handful of novel concepts. We will discuss our definition of following concepts in the following sections:

- **Dependency/Explanation Tree/Graph**
- **Rationalization**
- **Information Flow**
- **Source of Information $\approx$ Dependency**
- **Association**
- **Identifiability**
- **Compositionality** (for information extraction and generation)

2.2 Dependency/Explanation Trees

A **solved** information/data conversion problem is represented as an dependency/explanation tree shown in Figure 1. Each dependency/explanation tree represents a particular way of obtaining the output data using the information from input and memory only.

Note that a dependency/explanation tree can be a Directed Acyclic Graph (DAG) if an object serves as the source of information for more than one downstream objects, thus having two "parants" — but such objects can always be duplicated to reduce the DAG into a tree. A tree can also be converted into a DAG by merging corresponding nodes.

Thus, we can use the term "dependency/explanation tree" interchangeably with "dependency/explanation graph". But since graphs may mean something more general (e.g., a circular graph), we use the term "tree" for more specificity.

A dependency/explanation tree or DAG is almost equivalent to a program where each node is a variable and each edge is a an argument in a function call. The function call is the conversion function that we show in Figure 1. A conversion function is just a general function that can convert a set objects to another set of objects (See Section 2.7). Loops and recursions can be represented with some special control flow objects.

A dependency/explanation tree also represents an information flow from input and memory to output. It indicates where the information of output comes from.

As we will show in later sections, we can construct sub dependency trees for intermediate objects and they can become conversion functions by themselves. These conversion functions are "micro" explanations and can be reused compositionally to build larger explanations.

With a single training instance, there might be more than one valid ways (dependency trees) of obtaining the output. Yet not all of them have the same generalization performances. In order to maximize generalization performances, one should verify candidate dependency trees with multiple training instances and select the trees that work on all of them. In addition, one can also adopt some general heuristics to eliminate some candidate trees (e.g., by preferring some conversions over some others).

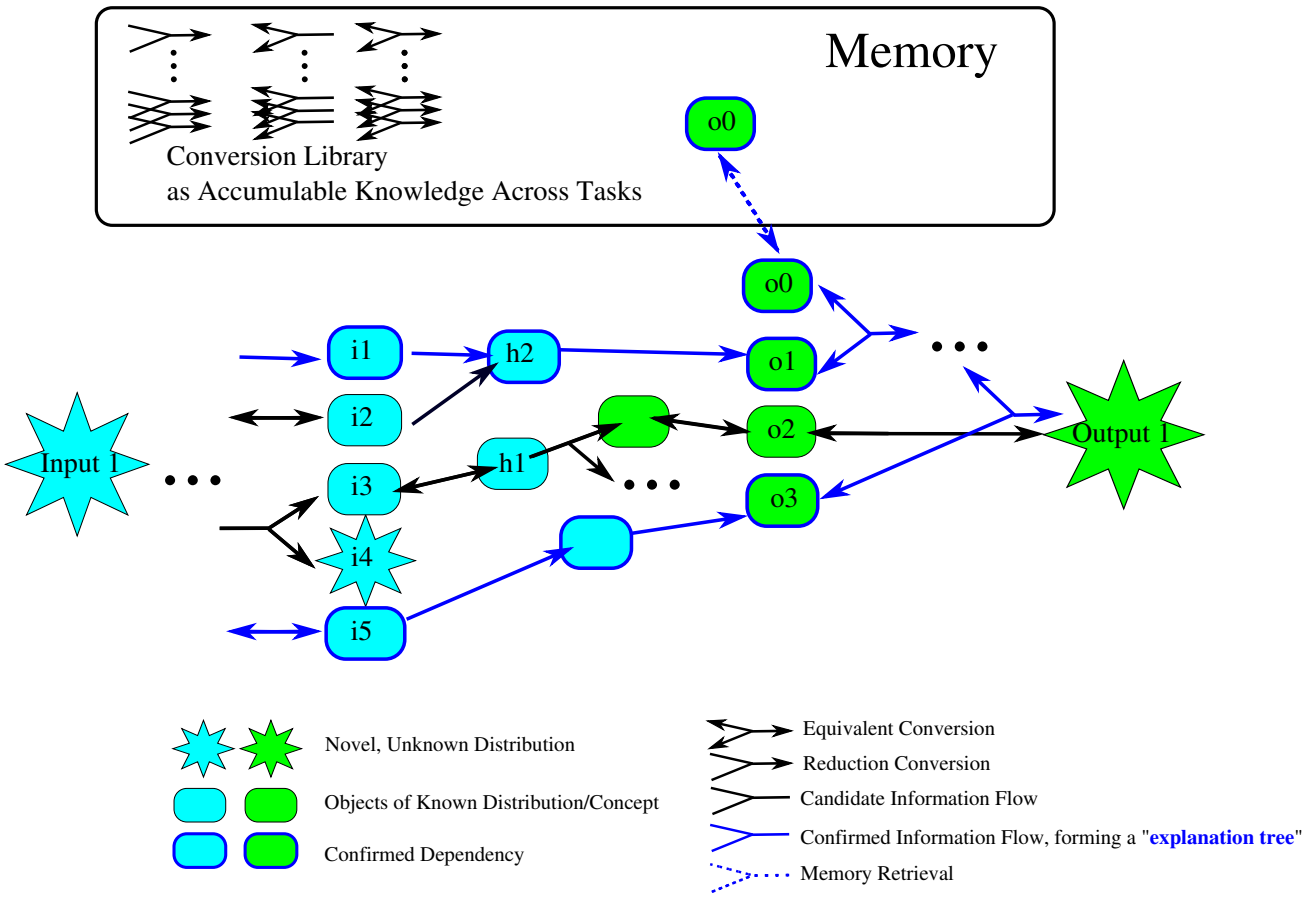


Figure 1: General Problem Solving as Information/Data Conversion. This figure shows a **solved** data conversion between input and output. The algorithm finds at least one arrow to the output that is "fully explained" — all of its dependencies are satisfied end-to-end, traceable to either input or memory, forming a **explanation tree**. All edges and nodes in this tree are **identifiable** either by its predecessor or its own distribution such that the same tree can be constructed from input and memory alone. There could be multiple explanation trees, each corresponds to one candidate interpretation of the data. Explanation trees are verified across all training examples (as shown in Figure 2, along with more details how a solution is obtained). Note that the dependency/explanation tree can be a Directed Acyclic Graph (DAG) if an object serves as the source of information for more than one downstream objects, thus having two "parants" — but such objects can always be duplicated to reduce the DAG into a tree.

2.3 Rationalization: Identifying the Source of Information

Rationalization is a unique human trait and we have strong internal urges to build theories/explanations for everything we observe. The evolutionary reason behind rationalization is that it helps us identify the reason, explanation and the source of information for any observed object. We would like to know why they are there, what cause them to exist and what induces their properties. We believe everything must have a reason, a source, a dependency or a cause. If there is not, we try to assign one.

We argue that this behavior is crucial for general problem solving — **rationalization is basically an constant-running general problem solving algorithm in our brain**. For every goal object, we will try to identify all immediate dependencies of it (i.e., where the information come from). We will find dependencies of dependencies until we reach something "solid" in the input or memory that is **identifiable** (discussed in Section 2.5).

This is done by first finding objects that are **associated** in some way to the object of interest and then check if there is any confirmed **information flow** in this **association**, as we will discuss in the following section.

2.4 Associations: Candidates for Information Flow

To identify the information source of an object, we usually need to consider multiple candidates. The candidate sources of the information of an object are the objects that are related/associated with this object in some way.

There could be many ways obtaining these associations, either from reasoning or just some intuitions. Once an association is identified, there may be a few instances of the same association. For example, if there is a green ball on the left of every red ball and there are 5 pairs of them in total. Then the association is "object on the left" and the number of instances is 5.

Instances of the same association can be used to verify the generality of the information flow. Dependencies/explanations that do not satisfy all instances can be quickly dismissed.

Every association can contain information flows. The process of checking whether there is any information flow in a particular association is a recursive process, illustrated in Figure 2.

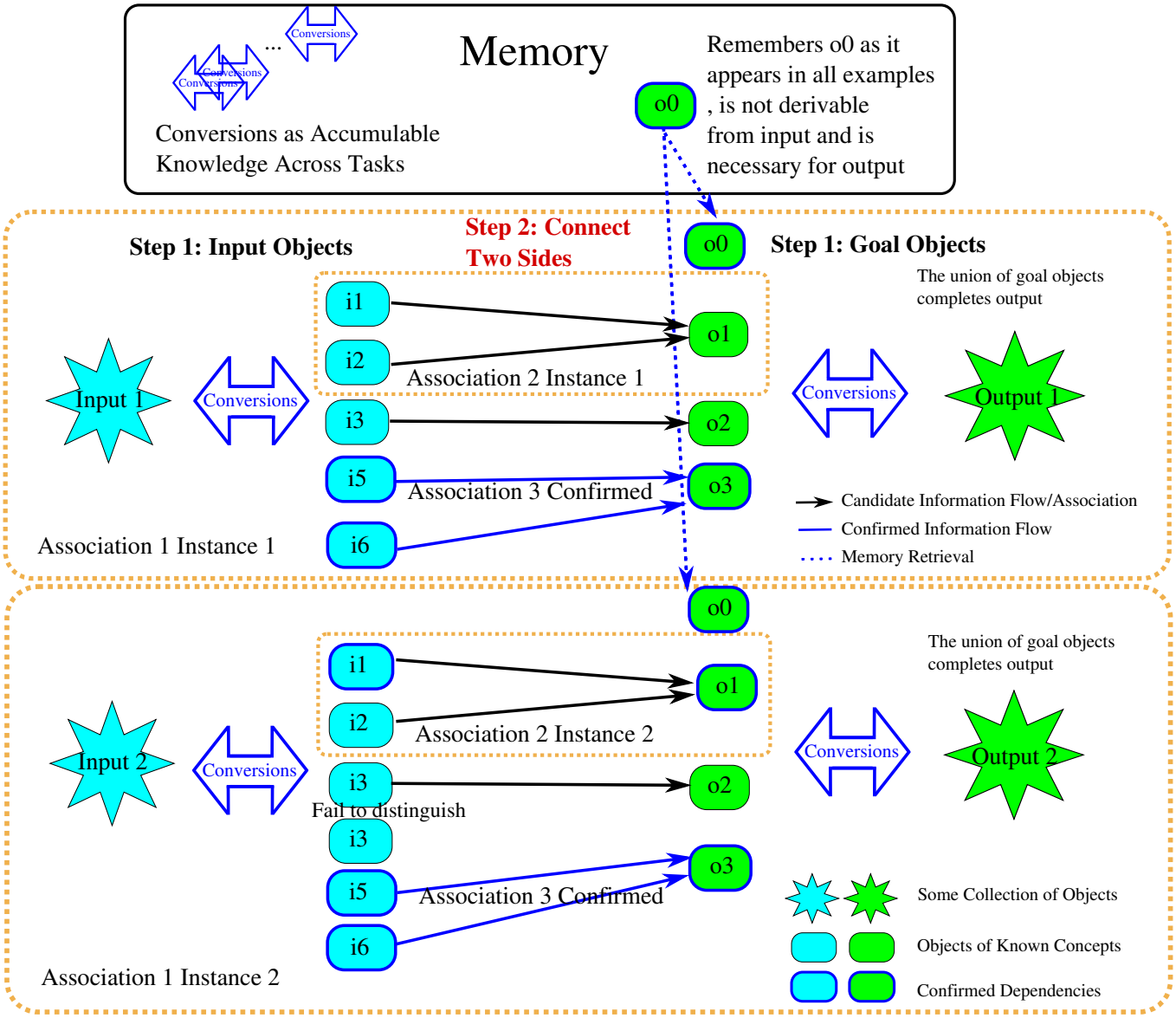
This formulation of association and instances is very general. In some sense, training input and output can be thought of as an association. Problem solving is just the process of finding confirmed information flow in this association. We do not even need to write a training script for the model since association and information flow finding is almost a constant-running "background" algorithm in the system. Merely presenting training input output side by side would make the model realize the relationship.

2.5 Information Identifiability

Narrow definition of identifiability: an object is **identifiable** if it has some unique properties to distinguish itself from all other objects. This only applies to existing objects.

For example, if we have a number red balls and one green ball in a bag. We say the green ball is identifiable and the red balls are not identifiable. But if one of the red balls has some marks on it or is noticeably distorted in shape, it become identifiable again.

Identifiability can actually transfer through data structures. For example, say there are three people female (165cm), male (175cm) and male(185cm) and a list [160,165,170,175,180,185]. We can say that the information



Check if an Association Leads to Confirmed Information Flow

Figure 2: Check if an association has confirmed information flows. We define the following recursive procedure: First conversions are applied to both input and output to decompose them as much as possible into fine-grained objects (as required by compositionality discussed in Section 2.6). Then new associations are detected between them. If the association is simple enough — an identity mapping for example, the information flow can be directly confirmed. If the association is complicated, we can reapply this recursive procedure on this association. It will return some information flows (if any) in this association. Once an information flow is confirmed between some input and goal objects, the rest of the flow connecting input and output can be computed using inverse functions from the corresponding conversion functions. Note that input objects in a confirmed information flow need to be "identifiable" (See section 2.5 for definition). Sometimes a conversion can also receive both input and output objects at the same time in the arguments to compute some intermediate goals. It is not important to distinguish input and output/goal objects since as long as they are identifiable they can be generated from input and memory.

165cm is identifiable, since that is the height of the only female. Furthermore, the number 2 is identifiable since it is the index of 165 in the list (suppose indices starts from 1).

General definition of identifiability: an objects is **identifiable** if there is a way (via a combination of rules) to uniquely extract/derive it from all information available. Thus every identifiable object is associated with a **extraction procedure** that performs the above extraction and corresponds exactly to the **information flow** (or **dependencies tree**) that leads to this object.

With this definition, general problem solving becomes a process of "identifying" the output from the input.

2.6 Compositionality and Conversion Functions

For most real world situations (e.g., what we discussed in [Mhaskar et al., 2016]), we can assume compositional structures in the data in the form of:

$$I(x_1, ..., x_8) = h_3(h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4)), h_{22}(h_{13}(x_5, x_6), h_{14}(x_7, x_8))) \tag{1}$$

suppose I is the input and the desired output O has the form of:

$$O = g_3(g_1(x_1, x_5), g_2(x_6, h_{11}(x_1, x_2))) \tag{2}$$

As we see above, a key property of compositionality is that the information of the system is not evenly distributed (as in a fully-connected neural layer or a homogenous "soup"). And the information between different constituents/parts is not entangled with each other. Different components in the compositional function should be treated as separate factors. For example, variables x_1 through x_8 above can be thought of as different pieces of primitive information and should not be lumped together.

We should decompose the compositional data into finer grained constituents (e.g., $h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4))$, $h_{11}(x_1, x_2)$, or x_1 and x_5) to properly "expose" the disentangled information so that we can efficiently achieve our purpose of information conversion.

For the same reason, compositional generation functions like $g_1(.,.)$, $g_2(.,.)$ and $g_3(.,.)$ will be applied to these "exposed" information at different levels to generate the output.

This motivates maintaining conversion functions for both decomposing and generation of data compositionally.

2.7 Conversion Functions

As we show in Figure 2 and discussed above, conversion functions are an important type of accumulative knowledge maintained by the system and it plays the role of compositional analyses and generation of data.

There are generally two types of information conversions: **Reduction** and **Equivalence**.

2.7.1 Equivalent Conversions

In an equivalent conversion:

$$y_1, y_2, ... = E(I_1, x_2, x_3, ...) \tag{3}$$

The output set of data $y_1, y_2, ...$ together with system memory M has the same amount of information as $x_1, x_2, x_3, ...$ together with M. For every equivalent conversion, there is an inverse:

$$x_1, x_2, ... = E^{-1}(y_1, y_2, ...) \tag{4}$$

Optionally, the system memory M could be utilized by E and E^{-1} in both conversions .

Example equivalent conversions:

- The conversion from a circle in 2D into a point in 2D and a radius
- The conversion between 56 and $8 * 7$
- The conversion between a 2D line segment and two 2D points
- The conversion between a object detection bounding box and the locations of the four boundaries (or equivalently the diagonal line segment)
- The conversion between a point in 2D and a pair of numbers (x and y).

Note that often there are multiple possible interpretations of a piece of data. For example a pair of values can not only correspond to a point in 2D, but also correspond to the height and weight of a person. Thus there exist multiple possible equivalent conversions for each piece of data. Which specific conversion is useful in solving a problem is context dependent and need to be determined on a case-by-case basis.

This also shows that information-wise the height and width of a person is "equivalent" to a point in 2D. With this type of conversions, the system can in principle figure out how to plot many peoples' physical information in a 2D plot for visualization.

In real life, there are "roughly equivalent" conversions. For example, an autoencoder can encode the picture of a cat into a latent code. This code can reconstruct the picture of cat through the decoder. But depending on how well the autoencoder is trained, there are noises in the encoded code and reconstructed picture. Whether this conversion can be considered conceptually as an equivalent conversion depends on whether the user/specification accepts this level of approximation.

There are also "contextually equivalent" conversions. In some context as far as the problem is concerned, a picture of a cat can be considered as equivalent to a word of "cat". This may require temporary construction of some equivalences. We will study this more in future work.

2.7.2 Reduction Conversions (Unidirectional, Can be Lossy)

In addition to equivalent conversions, many types of data conversions are lossy in the sense that they selectively extract some information from the data and discard some other informations.

Example reduction conversions:

- an object detector that only detects target objects in the data and ignores the rest of the image.

- an image classifier or an attributes detector that only output corresponding labels of interest and do not retain other information in their outputs.

Almost all unidirectional conversions can be understood as reductions, because the amount of information cannot increase by going through a function. In the case of generative models, the conversion function takes in extra arguments from the memory (e.g., weights) of the system.

2.7.3 Apply Conversions to Input and Output

As shown in Figure 2, the first action we perform for checking an association is applying conversion functions on input and output to get finer grained (or constituent) objects.

It is necessary to convert the **output** of a problem using only **equivalent** conversions, since the goal is to be able to reconstruct the output.

Let O be the output of a particular training example. By applying a set of **equivalent** conversions C_e repeatedly k times until no more conversion can be done. We have a set of "explained" data $O_e = o_1, o_2, ..., o_n$ following some known distributions.

$$O_e = o_1, o_2, o_3, ..., o_n = C_e^k(O) \tag{5}$$

Since C_e is a set of equivalent conversions, one can reconstruct exactly O with O_e. In fact, usually a subset of O_e should suffice due to the redundancies in applied conversions.

The **input** of a problem can be understood with either **reduction** or **equivalent** conversions or both, since we only need to extract relevant information that is enough to construct the output.

Let I be the input of a particular training example. By applying a set of **equivalent** and **reduction** conversions C_{er} repeatedly j times until no more conversion can be done. We have a set of "explained" data $I_e = i_1, i_2, ..., i_m$ following some known distributions.

$$I_e = i_1, i_2, i_3, ..., i_m = C_{er}^j(I) \tag{6}$$

3 "Object-Oriented Programming": Planning with Object-Oriented Conversions

In the above section we introduce the general framework and algorithm for information/data conversion. To make it applicable to real-world tasks, we have to be more specific about what entities it will deal with. We conjecture that in most cases, humans reasons in an object-oriented way and we need to focus our operations mainly on objects. This involves objects and their properties, the spaces they occupy (if any) and the interactions among objects. This is inspired by our early work on Object-Oriented Deep Learning [Liao and Poggio, 2017].

The first step of considering any relationship or association (see Figure 2) is "making sense" and "explaining" the data. To be specific, we conjecture that the first step humans perform when encountering a problem is to breakdown as much as possible the input into a collection of discrete[2] objects that can be reasoned over. This

[2]usually means local in space

action is performed on both input and output.

Once the input and output are broken down into objects, both direct conversions and new associations can be discovered. We will try to figure out how to generate each "goal object" and usually the union of the goal objects will form the desired output. If some complicated new associations are discovered, the entire algorithm described here can be applied recursively on them.

Note that we not only parse the output into objects, but also parse the difference between output and input into objects. Since these objects, if generable, can directly combine with some inputs to form outputs.

We will maintain an output-sized "completion" mask to indicate the parts of output that are already generable from the input, keeping track of progress. Whenever some output objects are generated, we will cross them off from the mask.

3.1 The Definition of an Object

An object generally refers to **an entity with some properties/fields**, similar to its definition in object-oriented programming. For example, it can be a word "foobar", a data structure list [1,2,3] or a rectangle with some color at some location in an image.

A simple object like a string has less properties like only position and length, while a more complex object like a rectangle may have properties like position, length, width, area, orientation, size, color, to name a few.

The list of properties of an object is extendable by running **property-extracting functions** on it and save the results to some fields (as determined by the corresponding functions).

3.2 Object-Oriented Conversions

Some example conversions for objects are shown in Figure 3.

The **generation functions** or **generators** can generate objects using essential information (for example generating a list from all of its elements or generating a rectangle from a diagonal line, or from a center, width, height and orientation.). For each generation function, the system should also maintain a corresponding **recognition function** or a **recognizer** that converts the object back to the above essential information that characterizes it. These functions are specific to each class of object. The recognition and generation function are analogous to the encoder and decoder in a typical generative model setting.

3.3 Object-Oriented Associations

In addition to conversion functions, we also need object-oriented associations. In general associations indicate statistically higher chance of information flow. Associations can be generated in flexible ways. Common object-related associations are alignment, containment, same properties, spatial proximity, among others. Objects have associations are more likely to interact and possess information flows.

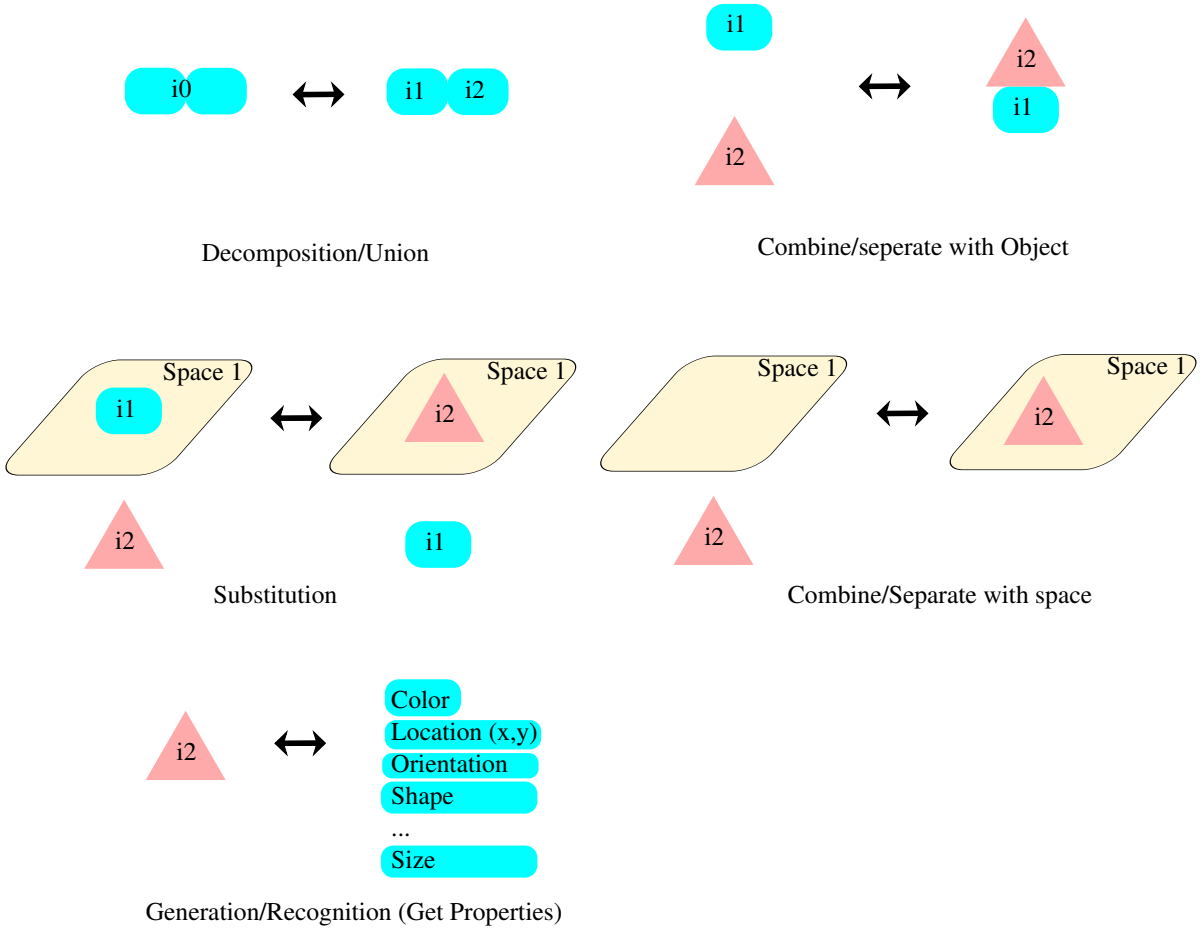


Figure 3: Example Object-Oriented Conversions. These conversions are concerned with properties related to objects, for example grouping, part-whole, aligning, overlay, space, attributes, generation, recognition, to name a few. Most conversion functions have inverse functions that supports converting in both directions.

4 Discussion

One naive approach to solve our problem is to randomly generate objects using conversion functions and hope to stumble across the desired output by large scale trials and errors. The problem of this approach is that the amount of computation can grow exponentially and it is clearly not how humans solve problems in general.

Our approach differs in the sense that we try to bridge the input and output by going both forward and backward (mostly backward). We try to "analyse" and "explain" the output by reasoning output objects together with input objects. Our reasoning is goal-driven, guided by information and associations induced by general theories of objects. Statistical machines may be trained to find better associations but it remains to be seen how much benefit it will yield.

References

[Liao et al., 2020] Liao, Q., Leonard, M., Pyo, B., Castillo, C., and Poggio, T. (2020). Universal format conversions.

[Liao and Poggio, 2017] Liao, Q. and Poggio, T. (10/2017). Object oriented deep learning.

[Liao and Poggio, 2020] Liao, Q. and Poggio, T. (2020). Flexible intelligence.

[Mhaskar et al., 2016] Mhaskar, H., Liao, Q., and Poggio, T. (2016). Learning real and boolean functions: When is deep better than shallow. *arXiv preprint arXiv:1603.00988*.

A Connecting Knowledge (Input and Memory) with Goals (Output) (OLD VERSION)

In step 1, we applied conversions to the input and output to get "explained" input and output data I_e and O_e.

Note that O_e is not yet fully explained in the sense that they are not supported by input I_e and memory M.

In some sense we have just finished the "parsing" stage. The step 2 corresponds to the "thinking" stage described in Section 2.

There are a few actions the system needs to perform in order to connect I_e and O_e.

A.1 Action 1: Finding Explanation Tree by Recursively Checking Explainability

We define a function $EXPLAINED()$ that checks if a node is "explained" and record possible explanations if **true**. Otherwise it returns **false**. It looks at all the conversions that leads to this node and for each conversion it recursively apply $EXPLAINED()$ on all of its inputs. If all inputs to this conversion are "explained", the conversion is explained. If any of the conversions to this node is "explained", this function returns true. Otherwise, it return false.

```
EXPLAINED(node)
 1 if Is Input or Memory Node then
 2     return True
 3 else
 4     conversions = node.conversions ;                    // conversions that lead to this node
 5     if conversions is EMPTY then
           /* it is a leaf but not input or a node in memory                              */
 6         return False
 7     hasExplanation = False
 8     node.explanations = list()
 9     for C in conversions do
           /* check all conversions that leads to this node                               */
10         conversionExplained = True
11         for inp in C.inputs do
               /* recursively checking if the input node is explained                     */
12             if EXPLAINED(inp) is False then
13                 conversionExplained = False
14         if conversionExplained is True then
               /* if all inputs of a conversion are explained                             */
15             node.explanations.append( C)
16             hasExplanation =True
17     if hasExplanation is True then
18         return True
19     else
20         return False
```

After checking explainability, one has a number of explanations in each node. We need to further confirm that each node can be generated in the forward direction by this conversion function without the output being present.

A.2 Action 2: Identify Correspondances and Merge Same Nodes

At this point, many pieces of data may directly correspond to each other in I_e and O_e. For every element in O_e, we can apply a search in I_e for essentially the same information. If we find a match, we can merge these nodes into one.

A.3 Action 3: Apply Higher-level Conversions

In principle I_e and O_e are already sufficiently expanded due to previous repeated application of conversions. Yet there is a possibility that some conversions are only useful for high level reasoning and are reserved for Step 2. For example, when solving a math task, all the theorems should be applied only in Step 2 since they do not really affect the understanding of the input and output.

For efficiency reasons, we should make most of the conversions triggerable by data instead of randomly applying them to all data.