Novelty in Goal-Oriented Machines Using a Thread Memory Structure

by

Saba Gul

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by .....
Patrick H. Winston
Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Novelty in Goal-Oriented Machines Using a Thread Memory Structure

by

Saba Gul

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Resourcefulness and creativity are desirable properties for an intelligent machine. The incredible adeptness of the human mind at seeing situations from diverse viewpoints allows it to conjure many techniques to accomplish the same goal, and hence recover elegantly when one method fails.

In the context of goal-oriented machines, this thesis presents a system that finds substitutes for the typical physical resource used to accomplish a goal, by *finding novel uses* for other, available resources—uses that these resources were not originally meant or designed for. In a domain where an object can serve multiple functions, this requires: (1) understanding the functional context the object is operating in; (2) building a realistic representation of the given objects, which often do not fall neatly into tightly-structured categorizations, but instead share properties with other 'boundary' objects. The system does this by learning from examples, and using the average member, or 'stereotype' as the class representative; (3) allowing imperfection: identifying properties that are not crucial for goal satisfaction, and selectively ignoring them; and (4) measuring similarity between objects to find the best substitute.

The system bootstraps with knowledge about the properties of the objects and is given positive and negative examples for the goal. It can infer, for example, that two objects such as an orange (the typical resource) and a ball (the positive example) are related in the context of finding a throwable object on account of their similarity in shape and size, but unrelated in the context of finding an ingredient for a fruit salad, because one is a fruit and the other is not. It then finds a substitute that shares shape and size features with the orange. If, on the other hand, we need an ingredient for a fruit salad, we can supply it another edible fruit as a positive example.

The system is implemented in Java; its performance is illustrated with 7 examples in the domain of everyday objects.

Thesis Supervisor: Patrick H. Winston
Title: Ford Professor of Artificial Intelligence and Computer Science

Acknowledgments

First and foremost, Patrick Winston, for his advice, support and guidance in patiently shaping and critiquing the ideas that resulted in this thesis.

Randall Davis and Marvin Minsky—their contagious passion for Artificial Intelligence is an inspiration.

George Verghese, for being a most helpful and accessible advisor and mentor for almost 6 years.

Members of the Genesis Group, especially Mark Finlayson, Jennifer Roberts, and Mike Klein, conversations with whom have guided this thesis.

My good friend Nada Amin, for her interest in my work. Some of the ideas that became this thesis were hatched in late-night conversations with her.

My brothers, for providing much-needed comic relief that kept me going.

My parents, without whose love and support I could never have accomplished this. It is to them that I dedicate this work.

Contents

List of Figures

Chapter 1

Introduction

If we are to build intelligent machines, they must be capable of understanding and accomplishing the same goal in multiple ways, and of being able to recover when the usual methods fail. In doing so, they must exhibit the human-like creativity and resourcefulness that allows us to operate in domains where the typical or usual physical resources for satisfying a goal are unavailable and substitutes must be sought.

The aim of this thesis is to take the first steps in imparting such resourcefulness to machines. Specifically, it aims to equip machines with a mechanism for inferring the correct context of comparison for two objects, and for finding substitutes by selectively ignoring object properties that are non-vital in that context. In doing so, it gives them the means to recognize when two objects can serve the same function by using one as inspiration, and *devising novel goals* for the other—goals that it was not originally intended or designed for.

The foundational methods used in the thesis mimic the kind of analogical reasoning that humans exhibit when comparing two physical objects. I use positive and negative examples to guide the system, and to infer the context in which the comparison must be made. I also use the concept of a stereotype—the most representative member of an object class and a realistic approximation to how humans categorize objects. This stereotype is calculated based on the examples the system has seen so far, and the frequency of observing the physical traits characteristic of the object. Along the way, I answer questions such as how to dynamically adjust our 'understanding' of objects in the domain, how to deal with diversity within an object class, and

how to treat examples that are exceptions to a class.

An example of the kind of problem the system can solve is the following. Given a plate as the inspirational or prototypical resource that normally satisfies the goal 'need an object to eat on', it can find a substitute for it in its absence, from a given set of available resources comprising of everyday objects: `(apple, chair, ball, book, shoes, frisbee)`. In doing so, it uses positive examples `(cup)` as well as negative examples `(car)` supplied to it. The output for this example appears in the figure below—the system offers a frisbee as an alternative to a plate.

```
The relevant threads are..

<thread>[plate, solid, round, concave, portable] [0, 0, 0, 0, 0, 0]</thread>

<thread>[cup, solid, concave, portable] [0, 0, 0, 0, 0]</thread>


The desirable properties for a substitute are:
solid
concave
portable


The best substitute for plate is: frisbee

<thread>[frisbee, inedible, frisbee] []</thread>

<thread>[frisbee, IS, solid, concave, small] [0, 0, 0, 0, 0]</thread>
```

Figure 1.1: Example of finding substitutes

The system is able to accurately find substitutes for missing resources in the domain of everyday objects. This is illustrated with 7 different examples.

1.1 Motivation by Example

In this thesis, I illustrate the usefulness of devising novel goals for an object with several examples. One such example will motivate the problem that the thesis addresses, and is imported from (Ulrich, 1988 [5]).

In his thesis, Ulrich focuses on computational design, contrasting two kinds of devices—those that exhibit the concept of 'function sharing', and those that do not. Function sharing, by his definition, is a mapping from more than one element in a schematic description to a single element in a physical description. In other words, it is the ability of a single functional element to serve multiple purposes. Devising novel uses for an object enables function sharing, which is hence an important consequence of this work. I motivate the thesis by discussing function sharing, showing that designs that display aspects of function sharing are more efficient and cost-effective than those that exhibit redundancy.

Ulrich illustrates the usefulness of function sharing using the examples shown in Figure 1.2. The diagrams are reproduced from his thesis.

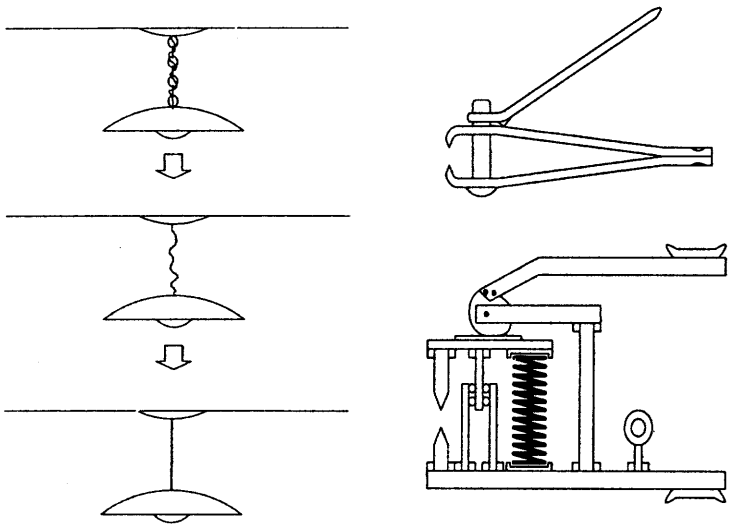

Figure 1.2: Examples of function sharing

In the figure, the left panel shows a ceiling lamp that is suspended using a link-chain with an electric wire woven into it. By recognizing the tensile properties of the electric wire, one can eliminate the need for the link-chain in the lamp.
In the right panel, the device at the top is able to accomplish the same function as the one below, albeit with fewer parts and a much cleaner and simpler design. This is possible because many elements in the device at the top are performing multiple functions, eliminating the need for certain other elements.

Ulrich further illustrates the usefulness of function sharing with the example of automobiles. If automobiles were designed without function sharing in mind, they would be much larger, heavier, and more expensive. It is because many elements in an automobile are able to serve multiple functions that automobiles can be manufactured relatively inexpensively. For example, the sheet-metal body can perform many functions such as weather protection, aerodynamic faring, structural support, electrical ground etc.

1.2 Outline

This thesis is organized as follows. Chapter 2 gives useful background knowledge on the resourcefulness exhibited by humans, laying grounds for the importance of context. It also discusses previous work that this thesis is grounded in.

Chapter 3 provides details on implementation, serves as a guide to the basic methods and techniques used, and gives results.

Chapter 4 includes a brief discussion and discusses the main contributions of the system.

Chapter 2

Background

2.1 Humans are resourceful creatures

Humans think of objects around them in terms of the goals they serve. The representations we have for these objects are used to accomplish these goals. Every time we see an object we have not encountered before, we think: 'I wonder what that does'.

We understand not only what purpose an object can serve, but also why other objects can or cannot serve the same purpose. This knowledge affords us debugging skills in situations that are unfamiliar, or where usual methods fail. This same knowledge enables us to devise novel uses for objects—an ability that is attributed to the resourcefulness, creativity or 'thinking outside the box' that humans possess.

One example of this resourcefulness that humans can exhibit was demonstrated by crewmembers of the NASA mission Apollo 13. Due to an oxygen tank explosion, the three astronauts had to use the Lunar Module as a 'lifeboat'. This necessitated the removal of carbon dioxide from the module, but the readily available, cube-shaped lithium hydroxide canisters built for purging carbon dioxide from the Command Module were incompatible with those used by the Lunar Module. With instructions from ground controllers, but using *only* materials available aboard the spaceship, the crew constructed the 'mailbox'—a jury-rigged arrangement that allowed them to use the canisters in the Lunar Module. In the absence of the prototypical resources, this

life-saving exercise involved using a sock as stuffing material, in addition to a plastic bag, a flight plan cover, and lots of duct tape.

This same resourcefulness or creativity is also exhibited when humans solve everyday problems, often as a natural and spontaneous process:

- At a free food event, a student is standing in line to get some food, and finds no more plates left to eat in. He looks at the frisbee he had just been throwing around, this time seeing it as a plate, and uses it to serve himself food.

- A runner sprinting down a hiking trail gets exhausted; he wishes to sit and rest. In the absence of any chairs or stools, he uses a large boulder.

Here, the plate and chair are the usual resources, or prototypes, for the goal at hand, whereas the substitutes used share certain physical and functional traits with them, serving as 'second best'. Being able to employ resources interchangeably to reach the same desired goal requires devising alternate uses for objects—uses that the object is not intended for. Much of this reasoning comes from using the prototype as inspiration, and making analogies with other, available resources.

Creativity in these examples requires finding inexact matches that are reasonably close to the required prototype, and *selectively ignoring* the missing properties. It is this knowledge of how much can be ignored, and what constitutes an acceptable substitute for a goal at hand, that my system is capable of inferring.

Thus, enabling machines to find substitutes for a given resource can be useful in situations where:

- The usual or prototypical resource(s) for a goal is (are) not available, such as in the Apollo 13 example.

- One wants to eliminate redundancy in design, by enabling the same design element to serve multiple functions. i.e. function sharing.

2.1.1 Resourcefulness requires understanding context

It is not enough to classify objects in the world based solely on their attributes or structural properties. Each representation for an object can answer a different question, but this representation does not operate in a vacuum—it is very closely linked to the goals the object can serve.

Thus, two objects can look similar for one purpose but very different for another. Consequently, classifying objects as similar or dissimilar is context-dependent i.e. the salience of object features or the physical traits we pay attention to depends on the context of the comparison, or the goal at hand. For example, if one is looking for a round, throwable object, an orange and a ball might be considered similar. But the same objects might be very dissimilar in the context of finding an ingredient for a fruit salad.

(Minsky 1991, [6]) provides another example of this phenomenon:

"...two armchairs of identical shape might seem equally comfortable as objects for sitting in, but these same chairs might seem very different for other purposes, for example, if they differ much in weight, fragility, cost, or appearance."

Because most everyday objects can serve multiple purposes, it becomes important to capture context in the learning technique the system uses. This information about context can then be exploited when searching for other objects that can serve the same purpose.

Thus, in order to find substitutes for an object, one must understand the context of the goal at hand.

2.2 Creativity and Novel Design

The concept of novelty or creativity is central to this thesis.

In his doctoral thesis, (Ulrich [5]) speaks of the concept of novelty in mechanical design. His system enables *function sharing*, or the ability of a

single functional element to serve multiple purposes. He hypothesizes that the unbiased application of the physical-feature based design operations in function sharing will automatically yield novel designs. His system produces designs that are viable candidates, but that are also, at times, unanticipated because the recognition procedures are more thorough and faster than human designers. More importantly though, they do not encode any prior intentions or functional information about the physical description that might produce a bias.

My work is, in part, inspired by that of Ulrich's. Even though his thesis is limited to design for mechanical devices, similar concepts of novelty recurr in my thesis.

I limit the notion of creativity to finding creative ways of using an object to satisfy a given goal. A creative use for an object is one for which it has not originally been intended. In my domain of everyday objects, each object has a *primary* use e.g. a chair is for sitting on, a cup is for drinking in, and so on. Secondary uses can be devised based on a subset of the objects' properties, that makes them acceptable candidates, or *substitutes* for that use e.g. eating in a frisbee because it is a solid, concave and portable object just like the prototypical resource—a plate.

2.3 The Concept of a Stereotype

In assuming different functions for the same object, humans use as inspiration the properties of the prototypical resource that is normally used to accomplish the goal. e.g. when looking for something to sit on, one uses the properties of a chair as inspiration. Henceforth, we shall call this the *inspirational resource.* But in a world where every object in a class is not identical, which object does one use for inspiration? More specifically, which properties can be inferred for the inspirational resource? Does one look at a chair that is black, or one that is white? A chair with a cushioned back or a wooden one?

Boundaries between object categorizations are not always distinct and complete. Cantor and Niedenthal 1984, [4], in their work on object categorization, revisit the concept of 'fuzzy' sets, first introduced by Lotfi A. Zadeh

in 1965. These sets are based on the *natural* categories that objects fall into, as opposed to the tightly-structured scientific taxonomies constructed by experts. For example, while a tomato is strictly a fruit, it is common to refer to it as a vegetable in conjuction with other ingredients that are vegetables. The authors persuasively argue that this natural and flexible categorization of objects is a departure from the classical approach that requires all members of a category to share a single, complete set of defining features, and which is characteristic of very few sets of objects.

The authors propose that despite the fuzzy structure of most natural categories, *prototypes* still "provide highly memorable reference points or standards of comparison against which objects in the environment can be compared in order to facilitate classification."[4]. This prototype (or stereotype) is the *average* member of its class, and is representative of the rest.

(Tversky 1977, [2]) proposes that a prototype is the member (or members) of a category with the highest summed similarity to all members of the category. (Rosch 1978, [14]) takes this concept a step further; she proposes that the more prototypical of a category a member is rated, the more attributes it has in common with other members of the category and *the fewer attributes in common with members of contrasting categories.* She says that a protoype exists to constrain, but in and of itself does not specify representation and process models.

(Vaina and Greenblatt 1979, [10]), in their seminal work that proposes a thread memory structure for object categorization, suggest that a the properties of the stereotype or prototype for a class be inferred from the frequency of encountering these properties. In their system, if a gray elephant is encountered more times then a brown one, it is concluded that the stereotypical elephant is gray.

In this thesis, I use the words stereotype and prototype interchangeably. My system uses the stereotype (for the inspirational resource) as inspiration for finding other objects that share properties, and can be good fits for satisfying the desired goal.

My system is a learner i.e. it uses past and new knowledge to make inferences and learn behaviors for future applications. In the context of inferring

stereotypes, this means that the stereotype for an object class will be dynamic. As more examples are given, the stereotype changes its property set based on the frequency of encountering a certain property.

2.4 Lattice Learning

This work uses inference methods similar to those used by (Klein 2008, [16]) in a technique called Lattice Learning, which is able to generalize accurately from a few examples. For instance, from the knowledge *'Robins, bees and helicopters can fly but cats, worms and boats cannot'*, it can learn the concept of flying objects and generalize it to birds, insects and aircrafts.

Lattice Learning uses negative and positive examples to learn concepts, representing object classes using a thread memory structure. It finds the most *general* type or classification such that the positive examples belong to it, but the negative ones do not. To do this, it walks down the types on each positive thread, from general to specific. If the current type is also found on a negative thread, it concludes that the type is not specific enough and keeps walking down the thread. If the type is not found on a negative thread, it concludes that it is at the right level of generality. Because a thread is required to follow a general-to-specific structure, the first type not found on a negative example is also the most general category for which the desired constraint is satisfied.

Figure 2.1 below illustrates Lattice Learning through a simple demonstration. In order to make the system learn about animate objects, it is given *cat* as a positive example and *chevrolet* as a negative example. The learning technique is able to infer from just these two examples, which of the remaining objects are animate and which are inanimate. It does this by generalizing, for example, that all mammals are animate, and hence dog and bat, which are mammals, must also be animate. Similarly, objects belonging to more general or more specific types of a class idenified as a negative example will also be tagged as not having the desired property. The power of Lattice Learning lies in its ability to generalize. It uses the examples to 'spread' the knowledge throughout the lattice in a magificent manner.

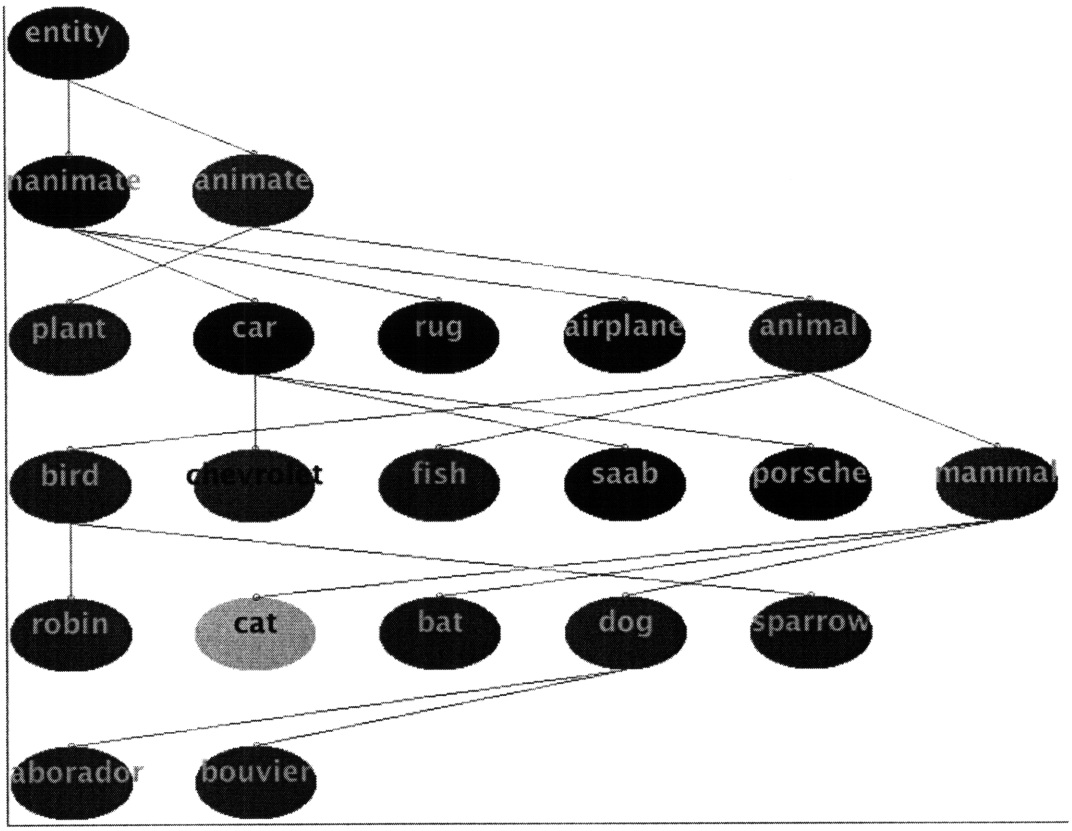

Figure 2.1: Lattice Learning Demo: Learning about animate objects

(Klein 2008, [16]) convincingly argues that it is much easier for the human mind to come up with specific examples than a general description of a class. While it may be tempting to argue that we could simply tell the system we are interested in animate objects, it is important to note that we can only do so if we know what this description should be. Usually, while a description is too complicated to explicitly name, it is easy to think of examples that are good fits. Lattice Learning is able to come up with classifications that match the given examples on-the-fly, as well as refine these with each new example.

My system is similar to Lattice Learning in the following ways:

- It generates descriptions for the desired goal on-the-fly, instead of requiring the specification of a goal description.

Just as classifications are hard to describe, so are goals. Specifying that you 'need something to poke a hole through paper' or 'an object that I can sit on' is hard not only for the user to specify, but also for the system to understand. The properties that are required for each of these goals would also be very hard to summarize. On the other hand, providing examples for each of these goals is much easier. Because the underlying representations for the example objects store their physical properties, inferences can be made about properties that are desirable, in order to find other good fits. Like any other learning system, more and better examples will improve the performance.

- It uses positive and negative examples to guide it, walking down the positive thread and comparing and contrasting properties with the negative threads. In addition, it has knowledge about the inspirational resource.

- Just like Lattice Learning looks for the most *general* type that has the desired classification, my system looks for the set of properties that is general enough that it does not unnecessarily limit the set of objects that can be used as substitutes, but specific enough that it does not include objects that are bad fits.

  As an example, say the system is looking for an object similar to a cup to drink in. It traverses down the thread of cup...

  `cup` → `solid` → `concave` → `cup`

  ...comparing it with the negative example elephant:

  `elephant` → `living thing` → `animal` → `mammal` → `elephant`

  `elephant` → `solid` → `elephant`

  It finds the attribute 'solid' on the negative example, and concludes that this attribute is not sufficient for the desired goal. Hence it keeps walking down the thread for cup.

The scope of my work, however, is distinct from that of Lattice Learning. While Lattice Learning deals with object classes and types in order to classify objects into categories, my system deals with sets of object properties to find functional substitutes for a given object.

In addition, while in Lattice Learning the object classes always follow a neat general-to-specific hierarchy, my system deals with property sets that might not follow this hierarchy. For example, the properties *light* and *concave* cannot be represented on a thread in a way that preserves the general-to-specific hierarchy, because neither property is more general or more specific than the other.

2.5 Inexact Matching

Often, the problem of finding creative ways to satisfy a goal boils down to deciding which inexact matches for an object will make acceptable solutions i.e what properties one can ignore, or 'make do' without.

(Minsky [11]) illustrates this very point, claiming that when humans think of objects and their properties, they often hallucinate about some of them. He gives the example of an apple, where in order to think 'apple!', we do not require that every imaginable property of an apple be satisfied. Instead, we have the ability to recognize that "if something is red and round, and has the right size and shape for an apple—and nothing else seems wrong", then it must be an apple. Similarly, if we hear a different rendition of a musical piece, we can still recognize it as the same piece with no problems whatsoever.

My system adds to this concept by hypothesizing that if a given object is being evaluated as a fit for a missing prototype, *which* of its properties one can ignore the absence of, and which of them one cannot, depends on the context of the goal. It is the context that determines the weight of each property; it is the context that gives us the required set of properties for satisfying a goal. For example, if we need a chair-like object to sit on, we will compromise on the color of the object, but not on its stability. However, if we are at a furniture shop and want to select a chair that is aesthetically suitable for a particular room, the color will now become an important property. Thus, this 'ignoring' is selective, and context-dependent.

Chapter 3

Implementation and Results

This chapter lays out the details of implementing the system, and analyzes the decisions made in doing so.

3.1 Basic guiding principles

I will start by laying out some basic principles that guide the implementation of this system.

- The system bootstraps with knowledge about the properties of the inspirational resource and negative and positive examples for the goal.
- The same node can be part of multiple threads, each representing a different context of operation for the node.
- The stereotype(s) for each object class is representative of its properties, and provides inspiration for finding new uses for related objects in the same or other classes.
- The heart of the system lies in determining the property set needed to satisfy a goal, and is done using examples provided to the system.
- The system is a learner—drawing inferences to create new knowledge, and learning from past knowledge. The more examples it is given, the better it is able to reason.

3.2 System Architecture

The system is implemented with 25 classes in Java, and builds upon the Gauntlet project developed by the Genesis Group at MIT CSAIL. The main goal of Gauntlet is to understand the computational nature of intelligence, particularly to understand the way that different human faculties (such as vision, motor and linguistic capabilities) come together to facilitate human intelligence. My system builds upon the basic infrastructure provided by Gauntlet; I therefore describe the basic representations below.

Gauntlet consists of a hierarchical system of Things (representative of objects), Bundles (a collection of threads) and Threads. These capture the essence of my reasoning sytem.

- A `Thing` is a data structure that represents an object in our world; it is comprised of Bundles.
- A `Bundle` is a collection of `Thread` objects that have the same key node i.e. the top node, which is a semantic representation of the object. Conceptually, this means that the two threads represent the same object. The bundle represents the entire collection of information we have for a particular object. All the bundles in the system are hashed by their keys and stored in a HashMap for easy lookup, as well as to avoid synchronization issues when bundles are updated as threads are added, deleted and modified.
- A `Thread` is a Vector object that stores the properties of an object in a linked manner, following the general-to-specific property hierarchy. We require a thread to begin and end on its key, and allow the same node to appear (either as a key or a regular node) on multiple threads.
- A `Property Vector` is a special kind of vector that stores properties for an object. Property vectors help store negative and positive examples, as well as non-hierarchical properties for an object. There are two relations a property vector can have. These are:

    1. the `IS` relation which indicates that an object has a certain property e.g. `(ball IS throwable)`

2. the `KIND-OF` relation which indicates that an object is an example of a particular class e.g. `(africansavannah KIND-OF elephant)`

The system takes a series of steps to find a substitute for a given inspirational resource. The flowchart below presents an overview of these steps. Each step will be explained in the sections that follow.

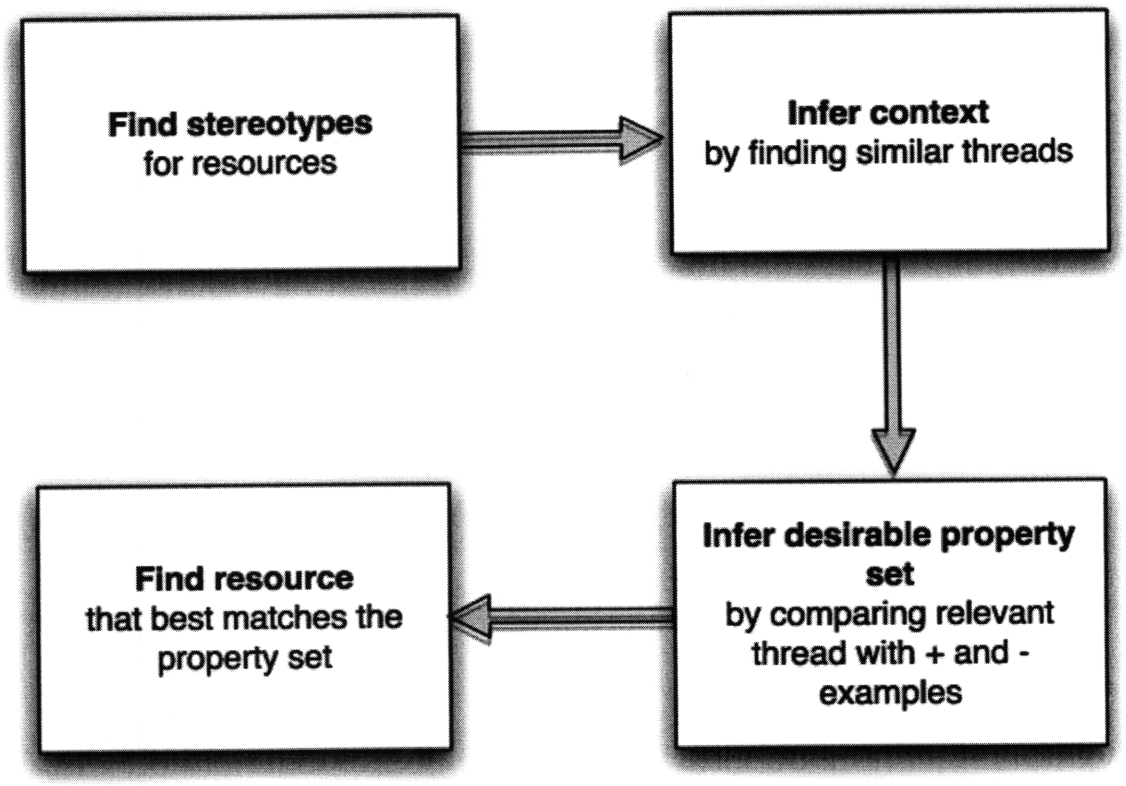

Figure 3.1: System Overview

3.3 Knowledge Representation: Capturing Context

The problem of representation is a focal point of Artificial Intelligence. The representation used determines the kind of reasoning the system is capable of; using a suitable representation can make a task much simpler. According to (Winston 1980, [7]), a good representation makes the important facts explicit and exposes constraint.

In this thesis, my knowledge representation of choice is a thread memory structure that is a variation of the thread memory first proposed by (Vaina and Greenblatt 1979,[10]). Their model consists of a loop-free, multi-link chain of semantic nodes, which represent the categories an object belongs to. For example, one can represent an elephant with the following thread:

`elephant` → `living thing` → `animal` → `mammal` → `elephant`

The thread is general-to-specific i.e. objects or properties that are downstream in the thread belong to more specific categories than objects or properties upstream. This organization makes information come out in the 'right order'; humans think of the most general classification of an object before thinking of more specific classifications.

Every thread begins and ends on its key. The key at the top is needed to be able to index the thread. This is also how the thread is stored i.e. it is hashed by its key. The key at the end is needed to complete the inference cycle. Because the key is the most specific class in the thread, putting it at the end allows us to infer, for example, that every elephant is a mammal.

My variation of this classical thread memory structure imposes the following additional constraint on a thread:

If X comes after Y in the thread, then all Xs are also Ys.

In other words, the properties of Y are a subset of the properties of X. This constraint can be imposed because of the general-to-specific rule that all threads follow. Let us revisit the elephant thread below:

`elephant` → `living thing` → `animal` → `mammal` → `elephant`

We see that this thread satisfies our constraint; it is easy to note that all animals are living things, and all mammals are animals. A thread such as the one below, however, does not satisfy our constraint:

`cup` → `light` → `concave` → `cup`

All concave objects are not light, hence this thread does not satisfy the constraint. To represent properties that cannot be represented in a manner that satisfies the constraint, we use property vectors which are not bound by it.

For example, one can represent the non-hierarchical properties of a ball using the following property vector:

`(ball IS inanimate light solid round)`

Thus, a big distinction between a thread and a property vector is whether the above-mentioned constraint is satisfied or not. Property vectors prove to be very useful for representing sets of properties that do not have a hierarchical relation with each other.

In this altered form, thread memory offers the following advantages:

- It lends itself very well to drawing inferences. For example, in the thread `boy` → `living thing` → `person` → `male` → `boy`, one can easily infer that *all boys are male.* This is different from saying that all males are boys. If instead we did not impose the additional constraint mentioned above, a thread such as `person` → `living thing` → `male` would be valid, but we would be incorrect in inferring that all living things are persons—they can also be plants.
- The properties of a thread memory simplify the implementation of context. Because of the existence of redundancy, each context of an object starts a new thread, and only information relevant to the context in

question is found on the same thread. This is explained in detail in the following section.

3.3.1 Thread Memory can capture context

In a thread memory structure, the same node can be part of multiple threads. This is a desirable property and allows the inference of context from the thread structure. Take the example of a boy:

`boy` → `person` → `male` → `boy [1]`

`boy` → `person` → `child` → `boy [2]`

`boy` → `person` → `student` → `boy [3]`

Here, [1] represents a boy in the context of properties that distinguish him from the female, [2] gives properties relevant to him being a child, and [3] tells us about the properties of a boy as a student. Subsequent classications of the boy that arise as a result of him being a male will reside on [1], those that represent his child-specic traits will be part of [2], and so on.

This is a helpful representation; if one is looking for properties of the person in the context of a child, threads [1] and [3] are irrelevant and need not be accessed at all. In a tree structure, as one traces up the path for boy, both relevant and irrelevant information is accessed and there is no way to distinguish between them.

The question that now arises is: when does one allow a separate thread to exist for the same node? In other words, why was a new thread created for the boy-child; how does one know that it represents a different context? The answer is that a new context is inferred if a new node does not lend itself to the strict general-to-specific hierarchy defined earlier whereby all nodes downstream must be subsets of those upstream. If this property cannot be preserved, the node is allowed to reside on a separate thread such that the property is preserved on both threads. For example, say child is added to thread [1] above:

boy → person → male → child → boy

The new thread implies that all children are male. If instead, child is added before male, it would imply that all males are children. Both these inferences are false. This is also how separate contexts are defined for threads; it allows one to infer that all nodes downstream from a node have similar properties or operate in similar contexts. In other words, the properties of a male will be, at the least, a subset of the properties of a boy.

3.3.2 Inferring Context from Positive Examples

How does the system infer the context of comparison from the set of physical resources it is given? For example, given an apple and an orange, how does it know in what context they are similar objects? In the context of their classification as edible fruits, or by virtue of their similar shapes and sizes?

The system infers context by comparing the inspirational resource with the positive examples it is given, exploiting the fact that different threads or property vectors are representative of different contexts. It looks to find a pair of threads—one from the inspirational resource and the other from the positive examples—such that they are most 'similar'. It then infers these threads and the properties residing on them to be the relevant context for comparison.

The measure of similarity is found by comparing each thread or property vector in the inspirational bundle to each thread or property vector in the positive bundle. This measure is weighted by the length of the threads. Pseudocode for this appears below.

```
//Method that returns the most similar thread-pair in 2 bundles

FIND_SIMILAR_THREADS (Bundle A, Bundle B)
   //compare each thread in A to each thread in B
   FOR each Thread T1 in A

      //remove the key and relational keywords
      IF A.IS_PROPERTYVECTOR THEN
```

```
      A.remove(''IS''))
      ELSE remove (A.KEY)
      ENDIF

         FOR each Thread T2 in B
            IF B.IS_PROPERTYVECTOR THEN
            B.remove(''IS''))
            ELSE remove (B.KEY)
            ENDIF

            //do string matching
               FOR each String S1 in T1
                  IF (T2.contains S1) THEN
                  matches ++
               ENDFOR

               //similarity measure, weighted by lengths of threads
               measure = (matches/A.length) * (matches/B.length)

               //update maximum measure and best pair so far
               IF measure > maxSoFar THEN
               maxSoFar = measure
               maxPair = <T1, T2>
         ENDFOR
   ENDFOR
//return the pair that has the highest similarity measure
return maxPair
```

Since threads have two keys (at the beginning and end) and property vectors only have one, a key is removed for the threads, so as not to give property vectors a comparison advantage. Relational keywords (such as IS) appearing in the property vectors are also removed.

The final similarity measure is weighted by the lengths of both threads; this gives more weight to a longer thread which has the same number of similar nodes as a shorter thread. e.g. Pair 1 below has a lower similarity score than Pair 2:

Pair 1:

```
ball → inanimate → ball
cup → inanimate → cup
```

Pair 2:

```
broccoli → inanimate → vegetable → broccoli
cauliflower → inanimate → vegetable → cauliflower
```

Using the technique outlined, let us consider an example of finding the right context between two bundles by finding the most similar thread. Say we are looking for an ingredient for a fruit salad, and the inspirational resource is an orange, whose bundle looks like this:

```
orange → inanimate → orange
orange → edible → fruit → orange
(orange IS solid round smooth-skinned small)
```

The positive example is a pineapple:

```
pineapple → inanimate → pineapple
pineapple → edible → fruit → pineapple
(pineapple IS solid cone-shaped rough-skinned large)
```

The relevant threads are found for the inspirational resource and the positive examples i.e. `orange` and `pineapple`. The similarity scores computed for each thread-pair in the bundles for orange and pineapple are shown in the table below:

Thread Pair	Similarity Score
edible→fruit→orange edible→fruit→pineapple	0.44
edible→fruit→orange inanimate→pineapple	0.0
edible→fruit→orange (pineapple solid cone-shaped rough-skinned large)	0.0
inanimate→orange edible→fruit→pineapple	0.0
inanimate→orange inanimate→pineapple	0.25
inanimate→orange (pineapple solid cone-shaped rough-skinned large)	0.0
(orange solid round smooth-skinned small) edible→fruit→pineapple	0.0
(orange solid round smooth-skinned small) inanimate→pineapple	0.0
(orange solid round smooth-skinned small) (pineapple solid cone-shaped rough-skinned large)	0.04

Figure 3.2: Calculating similarity scores

The most similar pair is thus:
orange → edible → fruit → orange
pineapple → edible → fruit → pineapple

The property vector for orange that tells us an orange is round and solid is deemed irrelevant, and the inferred context is that orange are pineapple

are related in the context of being edible fruits. The system gives the result shown below.

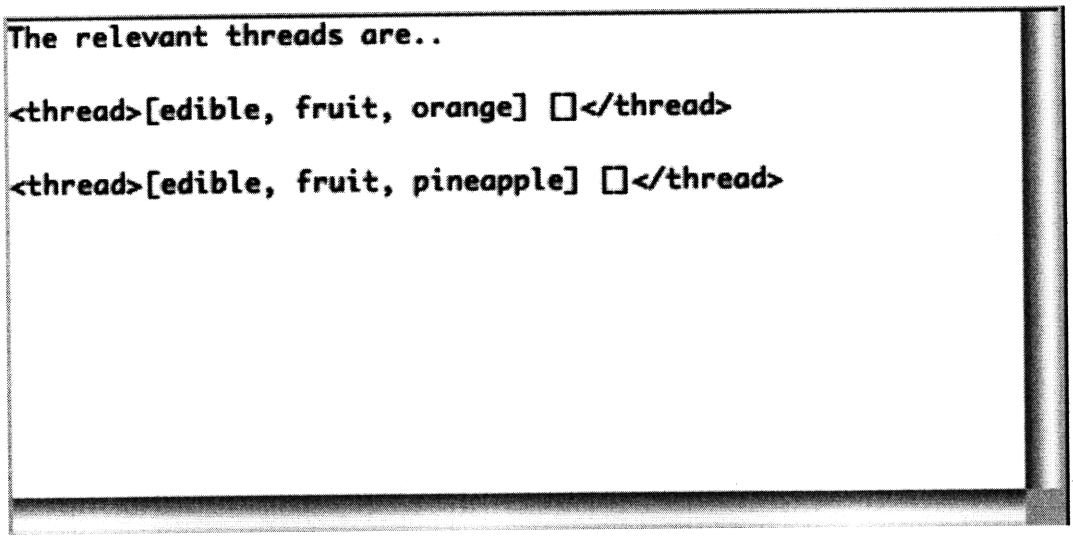

Figure 3.3: Finding the right context for orange and pineapple

Compare this to where, for the same inspirational resource i.e. orange, the positive example is a ball.

```
ball → inedible → ball
(ball IS solid round small)
```

In this case, the property vector for the ball that conveys its properties `(solid round small)` will be matched with the property vector for orange.

Thus we see that in one context, say that of finding a fruit for a fruit salad, one can give the positive example of a pineapple to allow the system to infer the correct context. In another context where the shape and/or size of the orange is of more consequence because we are looking for a throwable object, providing a similarly-shaped object such as a ball leads the system to the right conclusion, shown in the figure below.

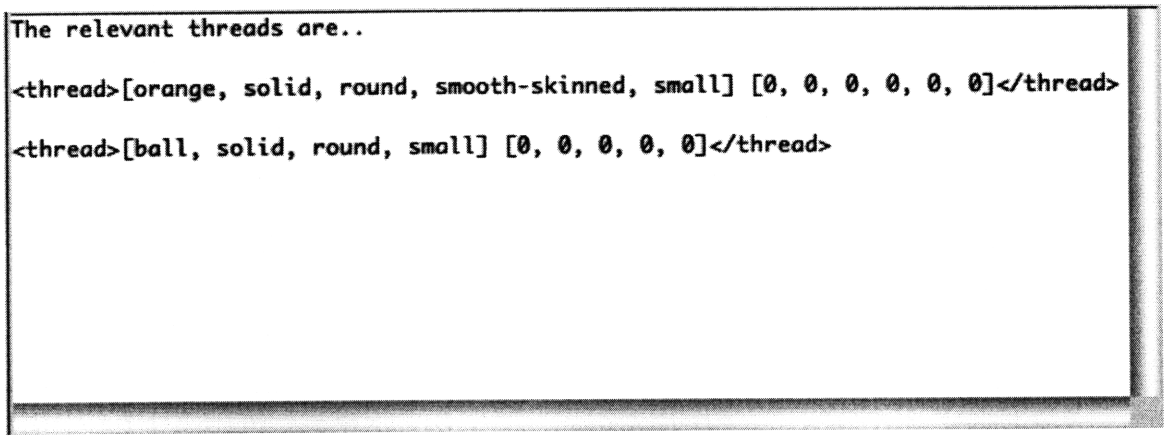

Figure 3.4: Inferring context for orange and ball

The underlying assumption here is, that if two objects are closely related in a context, the properties that are salient in this context will be more similar for the two objects than other properties.

3.4 All Knowledge is Not Created Equal: Absolute Versus Relative Knowledge

A distinction must be made between the two kinds of knowledge that the system acquires—that which is factual or absolute knowledge such as 'humans are living things' and that which is relative knowledge—true for a particular example but not necessarily for all objects in the same category e.g. 'Henna the elephant is blue' does not imply that all elephants are blue.

As examples are added for a particular object, the new knowledge they impart is generalized to superordinate classes. However, this generalization is selective, and applies only to threads, not property vectors. It is the threads for each object that store information that is strictly hierarchical in nature. In the threads, if a node X is downstream from a node Y, the inference that all Xs and Ys holds true. For example, let us consider the thread of an elephant again:

elephant → animal → mammal → elephant

This thread represents different levels of what the system recognizes to be factual knowledge. For example, that 'all mammals are animals', or 'all elephants are mammals'. This knowledge holds for all members of the elephant class.

It is these properties that the system generalizes or infers indiscriminately. This can be seen in the example below. The system is given these threads:

```
(car inanimate vehicle car)
(car IS solid wheeled)
(chevrolet KIND-OF car)
```

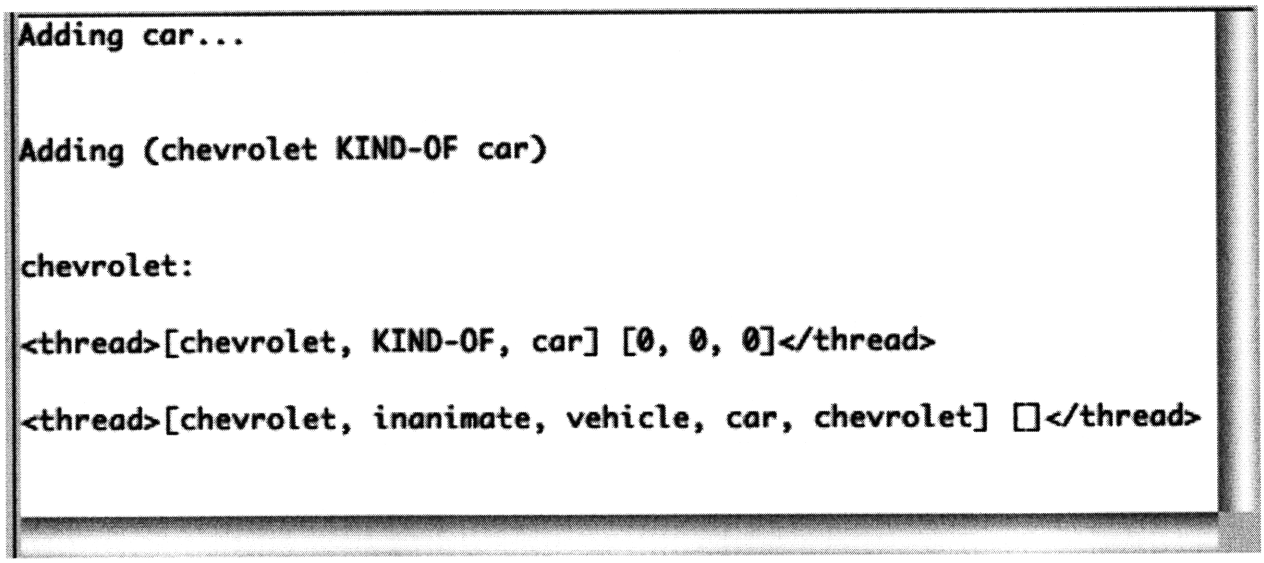

Figure 3.5: Adding an example for car

The system only infers the properties that are known to be true for all members of the `car` class, and that reside on the threads. No inferences are made from the property vector of `car`. Similarly, additional properties added to `chevrolet` will be generalized to add new properties for the `car` class.

3.4.1 Identifying stereotypes

The concept of stereotypes for a class allows objects within the class to be diverse, and for exceptions to the rules that constrain a class. When a class representative is needed the system finds the stereotype for the class, calculating this on-the-fly. Without a stereotype there is no way to know, when comparing say, a cat to a dog to a bat, which one is most representative of the mammal class, and can be used to generalize.

As explained in an earlier section, threads with the same key form a structure called a bundle. (Vaina and Greenblatt, 1979 [10]) suggest using the 'thickness' of the bundle in various parts to identify the stereotype through a method called identifying the forkpoint, which is the place in the two threads where they first diverge.

Consider the example of a girl:

`jane` → `girl` → `student` → `blonde` → `jane`

`helen` → `girl` → `student` → `brunette` → `helen`

So is our stereotype of a female student a blonde or a brunette? As the system is given more examples, one of the two threads will dominate. Thus, we can decide what the hair color of the stereotype is by looking at the thickness, or frequency of occurence, of the `blonde` and `brunette` bundles. This structure allows the stereotype to be a dynamic entity that changes with examples without requiring global updates.

In (Vaina and Greenblatt, 1979 [10]), the frequency of each observed property is stored in the following manner:

`girl` → `living thing [10]` → `person [10]` → `female [10]`

`girl` → `female [10]` → `student [6]`

The numbers in the parentheses are the observed frequency of each property as new examples are added, and capture the knowledge required to find the stereotype within the thread framework.

When devising novel uses for an object, one must draw on knowledge acquired from the stereotype for the class the object belongs to. The stereotype tells us what the common properties across the class are, and represents an image of the best example of the class. This best example is considered to be the average member of the class. Stereotypes can also be used to compare and contrast properties across classes.

The stereotype of a class does not preclude the existence of instances of that class that do not share certain features with the stereotype. The stereotype is built from the frequency of properties in instances of the class, rather than from analytical truths about it.

To calculate the stereotype, my system uses the same idea that (Vaina and Greenblatt 1979, [10]) suggest, using the frequency of occurence of a property to find the stereotype. However, in my architecture, different contexts lie on different threads. Hence, instead of finding the forkpoint, the system simply calculates the stereotype for an object by observing all the threads and property vectors in its bundle and picking the properties that have a higher-than-50% frequency of occurence. This includes all encountered examples of this object that are stored using the `KIND-OF` relation, as these are added to the bundle as well.

When threads are added, they have no associated thickness, as they comprise the absolute knowledge about the class, and are always added to the stereotype. We do not need to keep track of the frequency of encountering properties on the threads. It is the property vectors for which we store the frequency of each property on a separate vector. Thus each property vector is made up of 2 vectors—one that contains the properties and one that contains the frequencies. For example, say we have the following bundle for `chair`.

```
chair → inanimate → chair
(chair IS solid large low-COG) [0 0 0 0 0]
```

No examples of chairs have been encountered so far, and the frequencies are all 0. If no further examples of chairs are added, the stereotype for chair will be comprised of exactly the thread and property vector above.

Now suppose the system encounters a few kinds of chairs, as shown in the figure below.

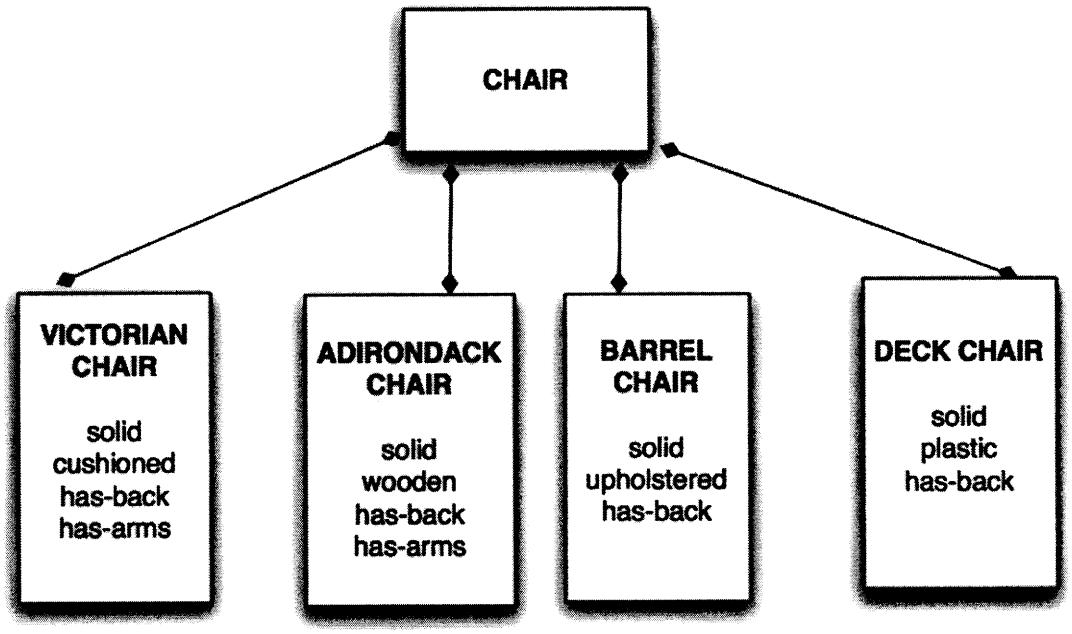

Figure 3.6: Types of chairs

These are added to the system using `KIND-OF` threads.

```
(Victorian-chair KIND-OF chair)
(Victorian-chair IS solid cushioned has-arms has-back)[0 0 0 0 0
0]

(Adirondack-chair KIND-OF chair)
(Adirondack-chair IS solid wooden has-arms has-back) [0 0 0 0 0 0]

(Barrel-chair KIND-OF chair)
(Barrel-chair IS solid upholstered has-back) [0 0 0 0 0]
```

```
(Deck-chair KIND-OF chair)
(Deck-chair IS solid plastic has-back) [0 0 0 0 0]
```

After each new example of a chair is added, the property vector(s) for chair get updated. Note that the chair bundle shown is just the updated version—the stereotype has not been calculated yet. The run is shown below:

```
Chair Bundle:
<thread>[chair, inanimate, chair] []</thread>
<thread>[chair, IS, solid, large, low-COG] [0, 0, 0, 0, 0]</thread>


Adding Victorian-chair...

Updated Chair bundle:
<thread>[chair, inanimate, chair] []</thread>
<thread>[chair, IS, solid, large, low-COG, cushioned, has-back, has-arms] [1, 0, 1, 0, 0, 1, 1, 1]</thread>


Adding Adirondack-chair...

Updated Chair bundle:
<thread>[chair, inanimate, chair] []</thread>
<thread>[chair, IS, solid, large, low-COG, cushioned, has-back, has-arms] [2, 0, 2, 0, 0, 1, 2, 2]</thread>
```

Figure 3.7: Adding new examples for the chair class

After all the examples have been added, the stereotype for chair is calculated to be:

```
<thread>[chair, inanimate, chair] []</thread>
<thread>[chair, IS, solid, has-back] [4, 0, 4, 4]</thread>
```

This is also shown in the figure below.

```
Chair

<thread>[chair, inanimate, chair] []</thread>

<thread>[chair, IS, solid, large, low-COG] [0, 0, 0, 0, 0]</thread>


Stereotype for chair

<thread>[chair, inanimate, chair] []</thread>

<thread>[chair, IS, solid, has-back] [4, 0, 3, 3]</thread>
```

Figure 3.8: Finding the stereotype for chair

Below I summarize some of the salient features of this method of finding a stereotype.

- Notice that some of the original properties of the chair bundle such as `(large low-COG)` have disappeared. As these appear on the property vector for chair and represent relative knowledge, if this knowledge is not corroborated by more than 50% of the examples, they will be overridden.

- This mechanism is able to mimic the human reasoning whereby we selectively ignore certain properties of an object that are not vital to it performing the function it is intended for.

  The examples of the `chair` set differ widely on their properties, and there is no clear majority for whether a chair is cushioned or whether it has arms. If a certain property is extremely diverse across the class, such that there is no clear majority, it is ignored and does not appear in the stereotype. For example, the properties that signify the material the chair is made of, such as `(wooden upholstered cushioned plastic)` are ignored. This is analogous to the human inference that the specificity of these properties is not vital to the object performing

its function. We can reason that while chairs may vary widely in how tall they are, what color, how much cushioning they provide, what pattern the fabric is etc, none of these properties are vital to the primary function of the chair—providing a surface to sit on.

This is based on the assumption that the examples provided will be reflective of the class. If all the examples consisted of cushioned chairs, the process will not work as expected. The hope is that over a large, representative dataset, properties that do have a significant majority *will* be of some functional value.

- This method of finding stereotypes is well-suited to dealing with exceptions i.e. objects that stand out or do not have properties that other class members do. An example is a bat, which belongs to the mammal class. While other mammal do not have the property of being able to fly, bats do. Because the frequency of the property `can-fly` on the thread for `bat` will be small compared to mammals that do not have this property, it will be ignored in the calculation of the stereotype for `mammal`. Hence, the bat's ability to fly will be rightfully treated as an exception in the class of mammals.
- If the bundle for an object class comprises of only threads (no property vectors), but property vectors appear in the examples for this object, these will be added to the bundle for the object. For example say we have an elephant bundle consisting solely of the following thread:

  `elephant` → `animal` → `mammal` → `elephant`

  Then we encounter an example of an elephant:

  ```
  (Henna KIND-OF elephant)
  (Henna IS large grey)
  ```

  This knowledge is generalized to create a new property vector for elephant, so the elephant bundle now looks like this:

```
elephant → animal → mammal → elephant
(elephant IS large grey) [1 0 1 1]
```

- The properties on the threads for an object always make it to its stereotype because they store absolute information. For example `<thread>[chair, inanimate, chair] []</thread>` carried through all the iterations of adding new kinds of chairs in Figure 3.8.

Thus when finding the stereotype for an object, the system aggregates both the absolute knowledge (threads) and the relative knowledge (property vectors), basing its inferences about the stereotype on the frequency of occurence of a property in the class.

Whenever further inferences need to be made, the system makes them from two kinds of properties:

- Those found on the main thread, that contain 'absolute' knowledge.
- Those found on the thread for the stereotype for the class, because this knowledge is representative of the class.

3.4.2 Inferring the Property Set

After the stereotypes for the resources have been found, and the most similar thread-pair is identified for the (stereotypes of) the inspirational resource and the positive example, the system has a set of properties of the inspirational resource, in the form of a thread or property vector, which comprise the relevant context.

Reproducing an earlier example, this relevant context looks like this:

```
orange → edible → fruit → orange
pineapple → edible → fruit → pineapple
```

From these threads (or property vectors), the system picks the one that belongs to the inspirational resource. It must now filter these properties to

infer the ones that are desirable in a substitute, using the given positive and negative examples.

How does the system use the positive and negative examples to filter these properties? It follows a simple algorithm that removes a property that appears in a majority of the negative examples, and does not appear in a majority of the positive examples. Majority here is defined as higher than 50%.

If we simply remove anything that appears on a negative thread, this could be problematic since it could be that the property is required, but is not sufficient. Take the example of the property `solid` which appears in a vast number of examples. It is a property that might be required, for example, for an object to be used to drink in, but it is not a sufficient property.

This is also true for properties that appear both on the positive and negative threads, in which case again, they could be required but not sufficient. This algorithm leaves both kinds of properties in the property set.

On the other hand, if a property is absent from a majority of positive examples, it is most likely not crucial for the goal at hand.

Thus the system *only* removes a property in two cases:

- The property appears in a majority of negative examples *and* is absent from a majority of positive examples.
- The property is absent in a majority of positive examples.

Below is pseudocode for finding the desired property set for a substitute:

```
// Finds property set with desirable set of properties for substitute

FIND_PROPERTY_SET (inspirationalKey, posExamples, negExamples)

   //find stereotypes for inspirational resource, + and - examples

   inspStereo = inspirationalKey.FIND_STEREOTYPE
   posStereo = posExamples.FIND_STEREOTYPE
   negStereo = negExamples.FIND_STEREOTYPE

   //find similar threads between inspiration and + example
   relevantThreads = FIND_SIMILAR_THREADS(inspStereo, posStereo)

   //get thread with relevant properties of inspirational resource
   propertySet = relevantThreads.FIRST_ELEMENT

   //for this thread, compare with + and - examples

      FOR each String s in propertySet

         //Find out if majority of the + and - examples contain property

         //if most - examples contain it AND most + examples do not
         IF ((negExamples.FIND_STEREOTYPE.MAJORITY_CONTAINS (s))
         AND (NOT (posExamples.FIND_STEREOTYPE.MAJORITY_CONTAINS (s))))

         //if most + examples do not contain it
         OR (NOT (posExamples.FIND_STEREOTYPE.MAJORITY_CONTAINS (s))))

         THEN propertySet.remove(s)

      ENDFOR

return propertySet
```

Using this property set, the system can now find the substitute from the given set of resources. It searches through the set of available resources, and

returns the object that contains the highest number of properties in the desired property set.

```
// Find object with desirable property and return it as the substitute

FIND_SUBSTITUTE (allResources, propertySet)

   // Remove + and - examples, since these cannot be a substitute
   allResources.REMOVE(posExamples, negExamples)

   //Find object that contains highest number of desired properties
   List SUBS = allResources.FIND_OBJECT_WITH_MAJORITY_PROPERTIES(propertySet)

return SUBS
```

Take the following example; the system is looking for a substitute for orange:

Inspirational Resource:
orange → inanimate → orange
orange → edible → fruit → orange
(orange IS solid round smooth-skinned small)

Positive Example:
apple → edible → fruit → apple
(apple IS round red small)

Negative Example:
car → inanimate → vehicle → car
(car IS solid wheeled large)

Other Resources:
pineapple → inanimate → pineapple
pineapple → edible → fruit → pineapple
(pineapple IS solid cone-shaped rough-skinned large)

```
ball → inedible → ball
(ball IS solid round small)

chair → inanimate → chair
(chair IS solid large)

broccoli → edible → vegetable → broccoli
(broccoli IS green leafy)

cup → inanimate → cup
(cup IS solid concave portable)
```

The result from the system appears below:

```
The relevant threads are..

<thread>[edible, fruit, orange] []</thread>

<thread>[edible, fruit, apple] []</thread>


The desirable properties for a substitute are:
edible
fruit



The best substitute for orange is: pineapple

<thread>[pineapple, edible, fruit, pineapple] []</thread>

<thread>[pineapple, inanimate, pineapple] []</thread>

<thread>[pineapple, IS, solid, cone-shaped, rough-skinned, large] [0, 0, 0, 0, 0, 0]</thread>
```

Figure 3.9: Finding a substitute for orange

The single best substitute is chosen from the available set of resources, *not* including the negative or positive examples. Because comparisons are made with the positive and negative examples to infer the desirable property

set, modifying these examples will modify the set of properties sought in the substitute. However, the system errs on the side of being conservative with negative examples. For example, suppose broccoli is added to the negative examples:

```
broccoli → edible → vegetable → broccoli
(broccoli IS green leafy)
```

The set of desirable properties still includes `edible` even though it appears on the negative example. This is because the system does not remove a property simply because it appears on a negative thread. If the same property that appeared on the negative thread were also absent from the positive examples, it would then be removed.

Now for this same example, suppose the positive example is changed to `ball` because the system is looking for a throwable object. The results change as shown in the figure below.

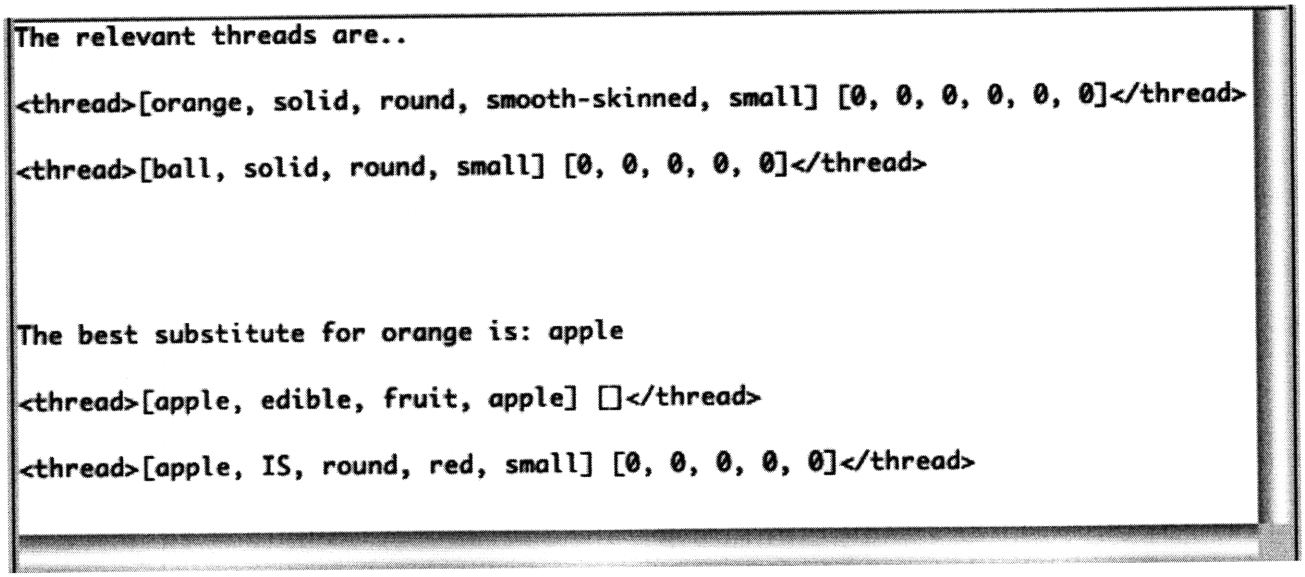

Figure 3.10: Finding a substitute for orange

This time the relevant context for the prototype changes from Pair 1, which was calculated when `pineapple` was the positive example, to Pair 2,

both shown below:

Pair 1:
`orange → edible → fruit → orange`
`pineapple → edible → fruit → pineapple`

Pair 2:
`(orange IS solid round smooth-skinned small).`
`(ball IS solid round small).`

In this context, the property vector of orange in Pair 2 is compared to all available resources, instead of the thread for orange in Pair 1. By this comparison, an apple, on account of having the properties `(round red small)` is a better fit than a pineapple which has the properties `(solid cone-shaped rough-skinned large)`. While both apple and pineapple are similar to an orange on account of being edible fruits, it is the positive example that determines the context and hence guides us in picking the apple instead of the pineapple. In this context, the fact that the apple is more of a throwable object due to its shape and size becomes more relevant than it being edible or a fruit.

Let us look at another example where we want a substitute for a plate:

`plate → inanimate → plate`
`plate IS solid round concave portable`

The positive example is:
`cup → inanimate → cup`
`(cup IS solid concave portable)`

The negative examples are:
`car → inanimate → car`
`(car IS solid large wheeled)`

`broccoli → inanimate → broccoli`
`broccoli → edible → vegetable → broccoli`
`(broccoli IS solid green)`

The rest of the resources are:

```
ball → inanimate → ball
(ball IS solid round small)

frisbee → inanimate → frisbee
(frisbee IS solid concave small)

apple → edible → apple
(apple IS solid round red small)

chair → inanimate → chair
(chair IS solid large)

car → inanimate → car
(chair IS solid large wheeled)
```

The results appear in the figure below.

```
The relevant threads are..

<thread>[plate, solid, round, concave, portable] [0, 0, 0, 0, 0, 0]</thread>

<thread>[cup, solid, concave, portable] [0, 0, 0, 0, 0]</thread>


The desirable properties for a substitute are:
solid
concave
portable


The best substitute for plate is: frisbee

<thread>[frisbee, inedible, frisbee] []</thread>

<thread>[frisbee, IS, solid, concave, small] [0, 0, 0, 0, 0]</thread>
```

Figure 3.11: Finding a substitute for plate

The relevant threads, with the highest similarity score are found to be:

```
(plate IS solid round concave portable)
(cup IS solid concave portable)
```

Notice that `solid` appears in a majority of negative examples, but since it also appears in the positive, it is kept in the property set of desirable properties for the substitute.

Most of the available resources are inanimate, but because the similarity score of the thread for cup that includes its property inanimate i.e. `cup` → `inanimate` → `cup` is lower than that of `(cup IS solid concave portable)`, the property inanimate is ignored.

The two closest matches are:

```
(frisbee IS solid concave small)
(frisbee IS solid concave small)
```

The positive example `cup`, because of its property `concave` helps narrow down the desirable properties and the system picks `frisbee` as the substitute.

Properties in the inspirational resource such as `round`, that do not appear in a majority of positive examples are removed from the property set. The substitute found does not need to have all or even most of the properties in the property set—it only needs to have more of these properties relative to the other available resources. In other words, it is the best possible match.

The system is able to accurately predict the substitute for 7 other example runs shown in the figure below. Cases where the predictions were not accurate or scenarios where it runs into problems are outlined in the next chapter.

Inspirational resource/ + examples/ - examples	**Other resources**	**Substitute**
ORANGE/apple/car	apple, ball, chair, broccoli, cup	PINEAPPLE
ORANGE/ball/car	pineapple, ball, chair, broccoli, cup, apple	APPLE
PLATE/cup/car	ball, apple, frisbee, chair, broccoli	FRISBEE
CHAIR/table/apple	broccoli, frisbee, rock, lamp, shoes, keys, trashcan	ROCK
PAPER/envelope/broccoli	orange, plate, ball, chair, plate, keys, napkin	NAPKIN
CUP/glass/frisbee	apple, plate, mug, chair, ball, orange, trashcan	MUG
GIFT-WRAP/fabric/plate	newspaper, envelope, chair, napkin, trashcan	NEWSPAPER

Figure 3.12: List of example runs

Chapter 4

Discussion and Contributions

Given the right examples, the system can accurately predict the best substitute for an unavailable resource.

It runs into problems if:

- It finds conflicting properties in the inspirational resource and the positive example e.g. suppose the inspirational resource is a large object whereas the positive example is small. If 'large' also appears in the negative examples, it will simply not be added to the required property set, but if this conflict arises for most or all of the properties, an empty or extremely limited property set might be returned, in which case no available substitute will be a good match. The system then returns with the information that no substitute is available.

- The relevant context resides on more than one thread or property vector. Since it only picks the best thread or property vector for which the inspirational resource and the positive example have the highest number of similar nodes, it can ignore important properties on other threads for the inspirational resource.

- There is more than one viable substitute. The system returns only one substitute—the first one that is the best match. This might be problematic in situations where there are multiple equally-fit substitutes, or where there is very little difference between two substitutes. In this

case it would be useful to return an ordered set of all possible substitutes.

- Picking the best substitute requires common sense. It is important to note that in many situations, resourcefulness also requires common sense knowledge about the domain. For example, a trash can is a solid, concave object just like a cup. The similarity in properties might lead the system to believe that one can use a trash can to drink in. It is common sense that would tell it one cannot drink from a trash can because it is unclean. Unless we are willing to accept results to problems that are useless because they defy common sense, we must incorporate some common sense into the system.
  Common sense can also give more creative solutions to problems at times. For example, knowing that a convex object can become concave if turned upside down expands the set of objects one can use to drink in, and offers novel alternatives in a resource-constrained environment.

In this thesis, I have made the following contributions:

- I have implemented a system that can make more accurate object categorizations, using the concept of stereotypes. This also allows for exceptions within a class. Giving more weight to examples that have a higher frequency of occurence is a more realistic approach to object classification, and captures human reasoning more closely. It is also a more accurate approximation of the world, where objects do not fall neatly into well-bounded sets.

- I have devised a simple mechanism that works well to *infer* the context of comparison, using positive examples. Incorporating context into the search for the best possible fit allows different properties of an object to become relevant for different functions. This is especially useful in a domain where the same object can operate in different functional contexts.

- I have built a system that can differentiate between absolute and relative knowledge for an object class, thus allowing for a more accurate and nuanced inference scheme that does not generalize indiscriminately.

- I have illustrated the ability of my system to accurately find substitutes for unavailable resources, bootstrapping with knowledge about the properties of the inspirational resource, the positive examples and the negative examples.

Bibliography

[1] Open Mind Common Sense [Online Resource]. Available: http://openmind.media.mit.edu/

[2] A. Tversky, *Features of similiarity*, Psychological Review, 84 (4), pp. 327—352, 1977.

[3] D. Gentner, K. Kurtz, *Relations, Objects and the Composition of Analogies*, Cognitive Science: A Multidisciplinary Journal, Volume 30, Issue 4, pp. 609—642, July 2006.

[4] N. Cantor, P.M.Niedenthal, *Making use of social prototypes: From fuzzy concepts to firm decisions.* Fuzzy Sets and Systems, 14, pp. 5—27, 1984.

[5] K. Ulrich, *Computation and Pre-Parametric Design*, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[6] M. Minsky, *Logical versus analogical or symbolic versus connectionist or neat versus scruffy*, AI Magazine, Volume 12, Issue 2, pp. 34—51, 1991.

[7] P. H. Winston, *Learning and reasoning by analogy*, Communications of the ACM, Volume 23, Issue 12, pp. 689—703, 1980.

[8] P. H. Winston, *Learning Structural Descriptions from Examples*, Massachusetts Institute of Technology, Cambridge, MA, 1970.

[9] K. D. Forbus, D. Gentner, 1994. *Mac/fac: A model of similarity-based retrieval.* Cognitive Science, 19, pp. 141—205, 1994.

[10] R. D. Greenblatt, L. M. Vaina. *The use of thread memory in amnesic aphasia and concept learning.* AI working paper, MIT Articial Intelligence Laboratory, 1979.

[11] M. Minsky, *The Society of Mind*, pp. 204—206, 1986.

[12] M.Minsky, *The Emotion Machine*, 2007.

[13] E. Rosch. *Cognitive representations of semantic categories*, Journal of Experimental Psychology: General, 194. pp.192—233, 1975.

[14] E. Rosch. *Principles of Categorization.* New Jersey: Lawrence Erlbaum, e. rosch and b. b. lloyd edition, 1978.

[15] O. L. Stamatoiu. *Learning Commonsense Categorical Knowledge in a Thread Memory System*, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[16] M. T. Klein. *Understanding English with Lattice Learning*, Massachusetts Institute of Technology, Cambridge, MA, 2008.

[17] A Kraft. *Learning, using examples, to translate phrases and sentences to meanings.* Massachusetts Institute of Technology, Cambridge, MA, 2007.

[18] M. Finlayson, P.H. Winston. *Analogical Retrieval via Intermediate Features: The Goldilocks Hypothesis*, Massachusetts Institute of Technology, Cambridge, MA, 2006.