Theo: A Framework for Self-Improving Systems

Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng,
Oren Etzioni, Marc Ringuette, Jeffrey C. Schlimmer

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

27 January 1989, 10:08

To appear as a chapter in "Architectures for Intelligence", K. VanLehn (ed.), Erlbaum, 1989.

Abstract

Theo is a software framework to support development of self-modifying problem solving systems. It provides a uniform representation in which beliefs are represented as values of slots of frames, and problems are represented by slot instances whose values are not yet known. Theo can also represent meta-beliefs and pose meta-problems about *any* of its beliefs or problems, including its meta-beliefs and meta-problems. This representation allows it to describe information about problem solving methods and strategies in terms of explicit beliefs about the problems for which these methods are appropriate. In addition, the meta-beliefs are indexed by the ground beliefs so that their retrieval cost does not grow as the size of the knowledge base increases.

This paper discusses the motivation and goals underlying the design of Theo, and provides an overview of the present incarnation of the system. We present experimental results showing the impact of three learning mechanisms currently implemented in Theo: caching of inferred beliefs, explanation-based learning of macro-methods for problem solving, and inductive inference of control information for guiding problem solving. We also discuss the relationship between Theo and two earlier frameworks with related goals: Soar and RLL.

This research is supported by the National Science Foundation under grant IRI-8740522, and by a grant from Digital Equipment Corporation.

Table of Contents

1. Introduction

The research reported here seeks to develop a general framework for learning and problem solving. Over the past two decades, research on machine learning has produced a number of mechanisms that address the question of how to generalize from examples (e.g., [Dietterich and Michalski 83, Mitchell, et al. 86, DeJong and Mooney 86]). This success has lead over the past few years to a number of attempts to construct self-improving problem solvers that employ these generalization mechanisms to improve their problem solving performance at various tasks (e.g., [Mitchell 83, Minton, et al 87, Laird, et al. 87]). Such attempts to construct self-improving systems raise important new research questions that go beyond the question of how to generalize from examples. A self-improving system certainly must address the issue of *how* to form general concepts from examples, but it must also address the issues of *which* concepts to learn, *when* to learn, *from what data and knowledge* to learn, *how to index* what it learns. It must be able to examine and modify most aspects of its own structure and processes in order to formulate and solve appropriate learning tasks at appropriate points in its development. In this light, the learning problem is inseparable from related issues of how the system is itself represented and what range of problem solving, reflection, indexing, and generalization mechanisms it can employ.

Theo is a framework for exploring this next level of research issues involving general problem solving, learning, and representation. As such, it draws on ideas developed in other recent research on architectures for learning and problem solving (e.g., [Laird, et al. 87], [Lenat 83], [Minton, et al 87]). The design of Theo is motivated by three overlapping goals, which are only partially satisfied at present. Theo is intended as a framework to support:

- *Basic research on general problem solving, learning and knowledge representation, and especially the interactions among these three issues.* Theo is a frame-oriented architecture that supports a particular approach to knowledge representation, provides a collection of inference methods, and includes mechanisms for generalizing from examples. This initial collection of system components can be modified and augmented in order to explore more specific architectures for problem solving, learning, and representation. For example, [Schlimmer 87] describes a search-oriented problem solver built on Theo.

- *Accretion of research results in a form that others may reuse and extend.* Theo is organized so that new representational features, new inference methods, and new learning mechanisms are each defined in terms of new slot definitions which can be incorporated by other users. Our hope is to move from the present era of throw-away research prototype software, to an era in which new software embodying new research results is more easily reused and research progress is more cumulative. For example, we have developed optional knowledge bases that define additional inference methods, learning mechanisms, and representation conventions.

- *An efficient framework for developing effective knowledge-based systems.* We seek to overcome one common failing of sophisticated frameworks for representation and inference: users often pay a high overhead in efficiency for a large set of system features, independent of which of these features are used in any individual program. Theo is intended to provide a range of sophistication and efficiency depending on the user's needs. For example, in defining new slots the user may specify which of Theo's inference methods should be used to infer values for instances of this slot, and whether or not Theo's learning mechanisms should be invoked for this slot. The user may also define his or her own inference procedures for inferring values of specific slots, encoding these either declaratively within the notation of Theo slot definitions (making them interpretable and modifiable by the system), or directly in Lisp code. This allows the user to choose between taking advantage of Theo's facilities for automatically refining inference methods, versus directly providing an efficient inference method encoded in Lisp.

In this paper we focus primarily on the first of these goals, and examine Theo as an architecture for self-improving problem solvers. The next section describes Theo and its representation, inference, and generalization components. Subsequent sections describe several ongoing research experiments conducted within Theo: experiments with explanation-based learning, inductive inference of control knowledge, and use of meta-reasoning about slot properties to guide inference. We conclude with a more general discussion of the relationship of Theo to other architectures such as Soar [Laird, et al. 87] and RLL [Greiner and Lenat 80].

2. Motivation and Overview of Theo

The organizing principles of an architecture may be usefully summarized at (at least!) two levels: functional and structural. A functional characterization describes the architecture in terms of features such as the range of beliefs that it can represent, the range of problems that can be posed to it, the range of inferences that it can draw, the range of knowledge that it can learn, the portion of its initial structure that is open to learning, and the computational complexity of its memory storage and retrieval mechanisms. A structural characterization describes the particular representation conventions and data structures, as well as mechanisms for inference, learning, memory storage, and retrieval. It is the structural characteristics that determine the functional properties of the architecture, although the derivation of one from the other may be far from obvious.

2.1. Organizing Principles

Below we summarize the structural organizing principles that underly the design of Theo. Following this, we consider the implications that these structural principles have on Theo's functional characteristics.

- Every *belief* in Theo is an assertion of the form (<entity> <slot>) = <value>, which represents the belief that some entity named <entity> stands in some relation named <slot> with another entity named <value>. For example, we might assert (fred wife) = wilma.
- Every *problem instance* in Theo is a pair of the form (<entity> <slot>), which represents the task of determining a justifiable belief of the form (<entity> <slot>) = <value>. For example, we might pose the problem "(fred wife)", whose solution is "wilma", or the problem "(current.chess.position best.next.move)". Note there is a one-to-one correspondence between beliefs and problems: each problem corresponds to a belief with a missing <value>, and each belief corresponds to the answer to some problem. Put another way, the token <slot> is the name of a relation, a belief is an instance of that relation, and a problem is a query that specifies a relation name as well as an element of its domain, and asks for the corresponding element of the relation's range.
- *Problem classes*, or sets of problem instances, are described either by a single token, <slot>, or by a pair of the form (<entity> <slot>). The form <slot> (e.g., wife) represents the class of problem instances of the form (?x <slot>) (e.g., (?x wife)), where ?x is any member of the domain of <slot> (e.g., if (wife domain) = male, then ?x may be any male). The form (<entity> <slot>) represents the class of problem instances of the form (?e <slot>) where ?e is a member of the class represented by <entity>. For example, (stone.age.male wife) represents the class of problem instances (?m wife), where ?m is some member of the set of stone.age.males. Simply put, (<entity> <slot>) represents the subset of the <slot> relation restricted to the subdomain specified by <entity>.
- Problem instances and problem classes are themselves entities. Thus, Theo can hold beliefs about problems and pose problems regarding problems just as it can for any other entity. For example, Theo could assert its belief that the problem of determining Fred's wife is difficult by asserting ((fred wife) difficult?) = t. Here (fred wife) is the <entity>, difficult? the <slot>, and t the <value> in the belief. For convenience, we drop the nested parentheses in

describing such beliefs, so that we simply write (fred wife difficult?) = t. Similarly, Theo's name for the corresponding problem of determining whether it is difficult to infer Fred's wife is (fred wife difficult?). Note we can easily state the belief that it is not difficult to determine whether it is difficult to determine Joe's wife: (fred wife difficult? difficult?) = nil.

- Theo stores all its knowledge about potential and recommended problem solving methods as beliefs about the corresponding problem class. For example, the belief (fred wife available.methods) = (inherits default.value) represents the assertion that the available methods for solving the problem (fred wife) are "inherits" and "default.value".
- Once a problem is solved, Theo may store the solution (i.e., assert the corresponding belief) in memory, indexed by the problem name. When it stores such beliefs, it also stores an *explanation* justifying the new assertion in terms of the beliefs on which it depends. Such explanations record dependencies among beliefs, so that when a belief is changed, Theo can remove dependent beliefs. This provides a simple forgetting mechanism for truth-maintenance.
- Explanations of beliefs also support a form of explanation-based learning which infers problem solving macro-methods from successfully solved problems, and stores these macro-methods as beliefs about the appropriately general problem class.
- An inductive learning method is used to acquire beliefs about the order in which available problem solving methods should be attempted for various problem classes.

The remainder of this paper explores some of the consequences of these organizing principles. In brief, the design of Theo is intended to provide three important features:

- **Highly uniform representation.** The one-to-one correspondence between beliefs and problems, and the fact that *every* problem is also an entity about which beliefs can be held, provide an architecture in which the same inference, learning, and forgetting methods that apply to problem solving in the task domain can also be applied to reasoning about Theo's problems, methods, capabilities, and explanations.
- **Broadly applicable learning mechanisms.** Theo currently has three learning mechanisms: caching of inferred beliefs, learning from explanations associated with successfully solved problems, and inductive inference for ordering problem solving methods. These mechanisms are intended to be applicable to all beliefs/problems describable by the system, so that in principle all parts of the system's knowledge and problem solving methods are open to learning.
- **Effective indexing of acquired knowledge.** Learning architectures must face the issue of effective indexing, or they face the problem that the more they learn the slower they become [Minton 88]. In Theo, problem names serve as indices to the problem definition, its solution, available problem solving methods, control information, explanations, and other properties of the problem. While we do not completely understand the implications of Theo's indexing mechanisms, it is clear that once the relevant information has been learned and cached for some problem, the time to retrieve this information is only weakly influenced by the number of irrelevant facts that are subsequently acquired about other problems.

2.2. An Introductory Example

This subsection presents a simple example of a typical Theo knowledge base, and illustrates Theo's inference and learning processes.

2.2.1. Representation

As described above, beliefs in Theo correspond to assertions of the form (<entity> <slot>) = <value>. Beliefs in Theo are stored in a frame-based representation, in which both frames and their slots correspond to entities.

Eleven simple Theo frames are depicted in Table 2-1. Each frame contains a list of slots and their values. Each sublist in the description of a frame is a slot whose value immediately follows the slot name, and whose subslots are listed after the value. Each frame and each of its slots is a Theo entity, and each entity's (sub)slots store Theo's currently explicit beliefs about that entity.

For example, the frame CUBE contains a slot called SPECIALIZATIONS, whose value is the list (CUBE1 CUBE2) (line 9). This illustrates how Theo stores the belief (CUBE SPECIALIZATIONS) = (CUBE1 CUBE2). Similarly, the frame CUBE contains a slot called HEIGHT whose value is presently unknown (as evidenced by the token "*NOVALUE*").

In general, we define an entity's *address* in the following fashion: If the entity is a top-level frame, its address is the name of the frame (e.g., CUBE). If the entity is a slot, its address is the sequence of slot names traversed to reach the slot from its top-level frame. For example, in Table 2-1, we say that the address of the HEIGHT slot of CUBE is (CUBE HEIGHT), and the address of the DEFINITIONS slot of the HEIGHT slot of CUBE is (CUBE HEIGHT DEFINITIONS). *The address of a slot is identical to the name of the Theo entity represented by that slot.* For example, (CUBE HEIGHT) is the slot address of the Theo entity named (CUBE HEIGHT) which represents the class of problems of determining the heights of cubes. The (sub)slots of (CUBE HEIGHT) represent Theo's beliefs about this problem class. In this case, the only explicitly held belief about this problem class is that (CUBE HEIGHT DEFINITIONS) = ((TH SIZE *DE*)).

Note that names of slots can also be names of top-level frames. For example, the top-level frame HEIGHT (lines 40-41) represents the class of problems of finding the height of any physical object. Theo may therefore infer beliefs about the problem class (CUBE HEIGHT) from its beliefs about the more general problem class HEIGHT. Furthermore, note that the frames VOLUME and WEIGHT also contain information defining additional slots (lines 31-38). Theo may use its beliefs about these two problem classes to infer beliefs about problem classes such as (CUBE WEIGHT) and (CUBE VOLUME), regardless of the fact that neither (CUBE WEIGHT) nor (CUBE VOLUME) is an explicitly represented slot in the entity CUBE.

2.2.2. Inference

The basic function of Theo is to access slot values (i.e., solve problems). If we query Theo regarding the value of (VOLUME GENERALIZATIONS) in the present example, it will return the stored value (PHYSOBJ.SLOT) (line 31). If we ask for the value of a slot which has no explicitly stored value, such as (CUBE1 WEIGHT), then Theo will attempt to infer a value based on its knowledge of the problem class (CUBE1 WEIGHT). All problem solving in Theo corresponds to inferring needed slot values.

A query for the value of (CUBE1 WEIGHT) leads to the sequence of subqueries shown in Table 2-2.[1]

---

[1] In fact, this is a highly selective trace showing only the slot accesses for the frames in Table 2-1. Many additional system-level slots are also accessed but are not shown in this introductory example.

Table 2-1: Fragment of a Theo Knowledge Base

```
 1 (physobj *novalue* (generalizations (frame))
 2    (specializations (box))
 3    (comment "prototypical physical object"))

 5 (box *novalue* (generalizations (physobj))
 6    (specializations (cube)))

 8 (cube *novalue* (generalizations (box))
 9    (specializations (cube1 cube2))
10    (height *novalue*
11       (definitions ((th size *de*))))
12    (length *novalue*
13       (definitions ((th size *de*))))
14    (width *novalue*
15       (definitions ((th size *de*)))))

17 (cube1 *novalue* (generalizations (cube))
18    (density 5)
19    (size 10))

21 (cube2 *novalue* (generalizations (cube))
22    (below cube1))

24 (physobj.slot *novalue* (generalizations (slot))
25    (specializations (density size height length width volume weight
26                      above below))
27    (domain physobj)
28    (available.methods (inherits defines drop.context use.inverse
29                        default.value)))

31 (volume *novalue* (generalizations (physobj.slot))
32    (domain box)
33    (range reals)
34    (definitions ((* (th height *de*) (th length *de*) (th width *de*)))))

36 (weight *novalue* (generalizations (physobj.slot))
37    (range reals)
38    (definitions ((* (th density *de*) (th volume *de*)))))

40 (height *novalue* (generalization (physobj.slot))
41    (range reals))

43 (above *novalue* (generalizations (physobj.slot))
44    (inverse below))

46 (below *novalue* (generalizations (physobj.slot))
47    (inverse above))
```

Lines in Table 2-2 such as line 3,

```
>5 tget (box weight)
```

indicate that as a subgoal of 4 other higher level slot accesses, Theo is attempting to get the value of the slot (BOX WEIGHT). Similarly, lines such as line 19,

```
<3 tget (5 (((cube1 density))))
```

indicate that Theo is returning from a level 3 get with a value of 5 for the slot (CUBE1 DENSITY). While the method for inferring slot values will be described in greater detail later, at this point it is sufficient to note that in order to determine the available methods for inferring the value of (CUBE1 WEIGHT) Theo retrieves the value of the subslot (CUBE1 WEIGHT AVAILABLE.METHODS). The value of this slot is itself inferred from the value of (PHYSOBJ.SLOT AVAILABLE.METHODS) (Table 2-1, lines 28-29). The retrieved list of available methods includes the items INHERITS, which suggests inheriting a value from a generalization of CUBE1, and DEFINES, which suggests applying a definition (such as the one in (WEIGHT DEFINITIONS) listed on line 38 of Table 2-1).

As shown in table 2-2, Theo attempts to infer the value of (CUBE1 WEIGHT) by first trying to inherit a value from (CUBE WEIGHT) (line 2), but this ultimately proves unfruitful.[2] Eventually (by line 18), Theo tries applying the DEFINES method, and this is ultimately successful.

The DEFINES method retrieves the value of (CUBE1 WEIGHT DEFINITIONS) and applies the definition in order to infer a value for (CUBE1 WEIGHT). Note that (CUBE1 WEIGHT DEFINITIONS) must itself be inferred, and it is obtained from (WEIGHT DEFINITIONS) in a series of slot accesses that are not included in the trace of table 2-2. Section 3.1 below discusses methods by which Theo can currently infer the value of slot DEFINITIONS on demand.

The retrieved value of (WEIGHT DEFINITIONS) indicates that the WEIGHT of some object, *DE*, can be inferred by multiplying the DENSITY of *DE* by the VOLUME of *DE*.[3] This leads Theo to attempt to retrieve the value of (CUBE1 DENSITY), as shown in Table 2-2 (line 18), and the value of (CUBE1 VOLUME) (line 20). Since this latter slot's value is not explicitly available, Theo backchains in an attempt to infer its value, and so forth. Table 2-2 summarizes the sequence of slot accesses performed to infer first (CUBE1 DENSITY) (lines 18-19), then (CUBE1 VOLUME) (lines 20-69), and finally (CUBE1 WEIGHT) (lines 1-70). Inference in Theo is typically of this fashion: methods are tried until one succeeds.

2.2.3. Caching and Explanations

Once a slot's value is inferred, the value may be cached (i.e., stored) in the slot, along with an explanation of how the value was inferred. More precisely, whenever Theo successfully infers a value for some slot, S, it examines the WHENTOCACHE slot of S to determine whether the value of S is to be cached. The default value for WHENTOCACHE (stored in the slot (WHENTOCACHE DEFAULT.VALUE)) is presently defined so that by default all slot values are cached[4].

When a slot's value is cached, the slot addresses that were used to infer this slot's value are recorded in its EXPLANATION subslot. Theo's explanation for the inferred value of (CUBE1 WEIGHT) is shown in Table 2-3. This explanation indicates that the value of (CUBE1 WEIGHT) is derivable from the definition

---

[2]Search up the inheritance hierarchy terminates at PHYSOBJ (line 4) because the domain of the WEIGHT slot is PHYSOBJ (inherited from the domain of PHYSOBJ.SLOT, Table 2-1, line 27), and thus WEIGHT is not defined for more general objects.

[3]The identifier *DE* refers to the entity whose slot value is to be computed. Slots can be viewed as relations defined over a particular domain and range, and *DE* refers to the domain element of the slot instance whose value is being inferred. Similarly, the identifier *RE* stands for the range element of the slot instance.

[4]We have not yet experimented in detail with policies for when to cache slot values. Our default policy of caching all slot values tends to consume many megabytes of memory, and we expect to eventually examine this issue in greater detail.

Table 2-2: Trace of Slot Accesses to Infer (CUBE1 WEIGHT)

```
 1 >1 tget (cube1 weight)
 2   >3 tget (cube weight)
 3     >5 tget (box weight)
 4       >7 tget (physobj weight)
 5         >9 tget (physobj density)
 6         <9 tget (*inferred.novalue* (((physobj density))))
 7       <7 tget (*inferred.novalue* (((physobj weight))))
 8       >7 tget (box density)
 9         >9 tget (physobj density)
10         <9 tget (*inferred.novalue* (((physobj density))))
11       <7 tget (*inferred.novalue* (((box density))))
12     <5 tget (*inferred.novalue* (((box weight))))
13     >5 tget (cube density)
14       >7 tget (box density)
15       <7 tget (*inferred.novalue* (((box density))))
16     <5 tget (*inferred.novalue* (((cube density))))
17   <3 tget (*inferred.novalue* (((cube weight))))
18   >3 tget (cube1 density)
19   <3 tget (5 (((cube1 density))))
20   >3 tget (cube1 volume)
21     >5 tget (cube volume)
22       >7 tget (box volume)
23         >9 tget (box height)
24           >11 tget (physobj height)
25           <11 tget (*inferred.novalue* (((physobj height))))
26         <9 tget (*inferred.novalue* (((box height))))
27       <7 tget (*inferred.novalue* (((box volume))))
28       >7 tget (cube height)
29         >9 tget (box height)
30         <9 tget (*inferred.novalue* (((box height))))
31         >9 tget (cube size)
             ...
36         <9 tget (*inferred.novalue* (((cube size))))
37       <7 tget (*inferred.novalue* (((cube height))))
38     <5 tget (*inferred.novalue* (((cube volume))))
39     >5 tget (cube1 height)
         ...
42       >7 tget (cube1 size)
43       <7 tget (10 (((cube1 size))))
44     <5 tget (10 (((cube1 height))))
45     >5 tget (cube1 length)
         ...
54       >7 tget (cube1 size)
55       <7 tget (10 (((cube1 size))))
56     <5 tget (10 (((cube1 length))))
57     >5 tget (cube1 width)
         ...
66       >7 tget (cube1 size)
67       <7 tget (10 (((cube1 size))))
68     <5 tget (10 (((cube1 width))))
69   <3 tget (1000 (((cube1 volume))))
70 <1 tget (5000 (((cube1 weight))))
```

of weight and the values of (CUBE1 VOLUME) and (CUBE1 DENSITY), that the value of (CUBE1

VOLUME) is derivable from the definition of volume and the values of (CUBE1 HEIGHT), (CUBE1 LENGTH), (CUBE1 WIDTH), and so on. In this case, all slot values were inferred via definitions using the DEFINES method. For example, "<--|weight.definitions.1|--" denotes the application of the first definition in (WEIGHT DEFINITIONS) (cf. Table 2-1, line 38). More generally, explanations can mention any of Theo's inference methods (e.g., INHERIT, DEFAULT.VALUE) in addition to slot definitions.

Table 2-3: Explanation of Inferred Value for (CUBE1 WEIGHT)

```
(cube1 weight) = 5000
  <--|weight.definitions.1|--
  (cube1 volume) = 1000
    <--|volume.definitions.1|--
    (cube1 height) = 10
      <--|cube.height.definitions.1|--
      (cube1 size) = 10
    (cube1 length) = 10
      <--|cube.length.definitions.1|--
      (cube1 size) = 10
    (cube1 width) = 10
      <--|cube.width.definitions.1|--
      (cube1 size) = 10
  (cube1 density) = 5
```

Saved explanations play two central roles in Theo. First, explanations are used for a simple form of truth maintenance. If any slot value in the system is changed, then the values of all dependent slots are uncached[5]. For example, if one changes the value of (CUBE1 DENSITY), then Theo will replace the value of (CUBE1 WEIGHT) with *NOVALUE*. If (CUBE1 WEIGHT) is subsequently queried, its value will be recomputed based on then current knowledge.

2.2.4. Learning

The second use of the EXPLANATION slot in Theo is to support a form of explanation-based learning. In particular, Theo composes the successful inference steps mentioned in the explanation, in order to form a special-purpose macro-method for subsequent use. Table 2-4 shows the specialized macro definition for WEIGHT which Theo derives from the explanation in Table 2-3. Note that the value of (CUBE1 WEIGHT EXPLANATION) provides both the information needed to construct the macro and to compute the macro's domain of applicability. By inferring the latter, Theo determines how far up the generalization/inheritance hierarchy it may store the new macro. In this case, the macro is stored in the slot (CUBE WEIGHT SPECIALIZED.DEFINITIONS) (despite the fact that the training example was related to the more specific CUBE1). This new SPECIALIZED.DEFINITIONS will be inherited when Theo attempts to compute the WEIGHT of other SPECIALIZATIONS of CUBE. Of course, the usual downside risks of forming macro-operators [Minton 88] apply here, since it may turn out that the macro-operator costs more time on the average than it saves. The impact of Theo's generalized macro learning is described and evaluated in greater detail in Section 3.

In addition to restructuring definitions in the knowledge base, Theo also improves its performance by

[5]In practice, the EXPLANATION slot is accompanied by its inverse, the DEPENDENTS slot. The latter contains a list of all slots whose values depend on this slot. When any slot's value changes, Theo uncaches that slot's DEPENDENTS.

Table 2-4: Specialized Definition of WEIGHT Inferred from (CUBE1 WEIGHT EXPLANATION) and Stored in (CUBE WEIGHT SPECIALIZED.DEFINITIONS)

```
(* (th density *de*)
     (* (th size *de*)
        (th size *de*)
        (th size *de*)))
```

modifying the *order* in which slot inference methods are attempted. By default, Theo attempts methods in the order in which they are listed in the AVAILABLE.METHODS slot. Of course this default ordering may not be optimal for all slot inferences, as is the case in the example of Table 2-2 where Theo attempts the unsuccessful INHERITS method before the DEFINES method (lines 2-17) and does more work than necessary to infer the value of (CUBE1 WEIGHT). In order to learn a better ordering for methods, Theo keeps a set of simple statistics on the cost and likelihood of success for each slot inference method. Table 2-5 lists the data Theo retains after inferring the value of (CUBE1 WEIGHT).

Table 2-5: Data collected while inferring (CUBE1 WEIGHT)

Computations	Odds of Success	Cost of Success	Cost of Failure
(WEIGHT INHERITS)	0/3	?	459K[6]
(WEIGHT |WEIGHT.DEFINITIONS.1|)	1/3	883K	113K
(WEIGHT DROP.CONTEXT)	0/3	?	0
(WEIGHT USE.INVERSE)	0/3	?	23K
(WEIGHT DEFAULT.VALUE)	0/4	?	39K

Theo utilizes the data from Table 2-5 to reorder the methods for WEIGHT, by moving DEFINES to the front of the list. If this advice could have been followed beforehand, the number of slot accesses shown in Table 2-2 to infer (CUBE1 WEIGHT) would have been reduced from 70 to 20.[7] Section 4 describes and evaluates this learning method in further detail.

The above example is intended to illustrate the style of knowledge representation, inference, and learning in Theo. Each of these topics is discussed in greater detail in subsequent sections. The primary points illustrated by the above example are:

- All explicitly held beliefs in Theo are stored as values of slots of entities.
- All problems in Theo are queries for the value of some slot of some entity.
- Slots of entities are themselves entities which represent the problem of inferring the value of that slot. As entities, these slots have subslots (e.g., AVAILABLE.METHODS, DOMAIN, DEFINITIONS) which store beliefs about how to infer a value for the given slot. If these

[6]Cost(Succeed) and Cost(Fail) are measured in terms of CPU time.

[7]In fact, reordering of methods can occur either in *guaranteed* mode, in which Theo does not reorder any methods until a statistically significant amount of evidence has been collected; or in *heuristic* mode, in which Theo reorders the methods without guarantees of statistical significance.

subslots themselves have no value, they may themselves be inferred on demand as for any other slot.

- It is useful to view Theo as an infinitely large *virtual* datastructure which is incrementally made explicit on demand. This virtual datastructure delimits the set of all beliefs which Theo can in principle represent, and therefore hold. It also delimits the set of problems which may be posed to Theo.
- When Theo is successful in inferring the value of some slot, it may cache the inferred slot value along with an explanation that justifies the value.
- Explanations of slot values are used to provide a straightforward form of truth maintenance, and to guide a form of explanation-based generalization.
- Theo currently employs three types of learning mechanisms: caching of slot values, inferring macro-methods from the explanations of inferred slot values, and inductively inferring the order in which slot inference methods should be attempted.

The following sections describe Theo's inference, representation, and learning mechanisms in greater detail. Following this, we consider the relation between Theo and two earlier systems: Soar and RLL.

3. Inference

Table 3-1: Summary of Theo's Layered Inference Mechanism

To infer (CUBE1 WEIGHT):

- Layer 1: Apply Lisp function in slot (CUBE1 WEIGHT TOGET) to the address (CUBE1 WEIGHT)
  - If unspecified, the default value of TOGET implements layer 2.
- Layer 2: Apply list of methods in (CUBE1 WEIGHT AVAILABLE.METHODS) to the address (CUBE1 WEIGHT) until a value is obtained.
  - If unspecified, the default value of available methods is: (DEFINES INHERITS DROP.CONTEXT DEFAULT.VALUE).
  - The DEFINES method implements layer 3.
- Layer 3: Interpret definitions found in (CUBE1 WEIGHT DEFINITIONS) and in (CUBE1 WEIGHT SPECIALIZED.DEFINITIONS).
  - Value of DEFINITIONS subslot may itself be inferred from knowledge of slot's INVERSE, PLURAL, SINGULAR, etc.
  - Value of SPECIALIZED.DEFINITIONS may be inferred via explanation-based learning.

As illustrated by the above example, Theo infers the value of a given slot S based on beliefs stored in the subslots of S. This subsection describes in greater detail Theo's mechanisms for inferring slot values.

There are three layers of discipline at which the user can specify the inference method for a particular problem class (e.g., slot). These are summarized in Table 3-1. At the most basic layer, each slot (i.e., problem class) has a subslot named TOGET, whose value is a Lisp function. This function takes as an argument the address of the slot whose value is to be determined, and returns that slot's value along with an explanation. The basic inference function in Theo, called TGET, simply applies this function to the address of the slot in question. Thus, the user can specify an arbitrary Lisp function to infer the value of

slot S simply by storing the name of the Lisp function in the slot (S TOGET).

If the user defines some new slot, S, and does not specify a value for (S TOGET), a value will be inferred for (S TOGET) from the system-provided value of (TOGET DEFAULT.VALUE). This default value of TOGET establishes the next higher layer of discipline for specifying inference methods. At this layer, inference methods are specified by the AVAILABLE.METHODS slot of S.[8] See, for example, the slot (PHYSOBJ.SLOT AVAILABLE.METHODS) from Table 2-1. Theo infers the value of S by sequentially attempting the methods specified in (S AVAILABLE.METHODS) until one succeeds in returning a value.[9] In fact, each method corresponds to the name of a (possibly virtual) subslot whose value is the result of applying that method to the current problem. Thus, in inferring the value for a slot such as (BOX1 WEIGHT), Theo may cache explicit values for various subslots such as (BOX1 WEIGHT DROP.CONTEXT), (BOX1 WEIGHT DEFAULT.VALUE), etc. Appendix I summarizes the legal entries in the list AVAILABLE.METHODS, as well as the definitions of the corresponding inference methods.

If the AVAILABLE.METHODS of the slot includes the token DEFINES, then Theo will attempt to utilize the DEFINITIONS and SPECIALIZED.DEFINITIONS[10] subslots to infer a value. DEFINITIONS provide the third, and highest level of discipline for specifying slot inference methods. The value of the DEFINITIONS slot is a list of expressions which are evaluated in sequence in an attempt to infer a value for the slot. For example, the slots (WEIGHT DEFINITIONS) and (VOLUME DEFINITIONS) in Table 2-1 illustrate the use of the DEFINITIONS slot.

For convenience in writing slot definitions, the variables *DE*, *RE*, and *ME* designate predefined slot addresses. *ME* designates the address of the slot being defined, *DE* designates the address of the domain element of this slot, and *RE* designates the address of the range element of this slot. For instance, if the value of the slot (CUBE1 VOLUME DEFINITIONS) refers to these variables, then *ME* will evaluate to the address (CUBE1 VOLUME), *DE* will evaluate to the address CUBE1, *RE* will evaluate to the address 1000,[11] (i.e., the range element, or value, associated with this slot (cf. Table 2-2, line 69)). These variables, together with the function TH,[12] provide a convenient means of specifying entity addresses relative to the address of the slot whose value is being inferred. See, for example, the definitions illustrated in Table 2-1.

The layered inference mechanism in Theo allows the user to associate efficient Lisp procedures with selected slots (by asserting a value for the TOGET slot of that slot), to utilize a list of system standard inference methods such as INHERITS and DEFAULT.VALUE for other slots, and to utilize definitions

---

[8]To be more accurate, layer 2 inference methods are in fact specified by a slot named METHODS, whose value is inferred from the value of the AVAILABLE.METHODS slot. However, it is easiest to think of METHODS as an implementation detail, and to think of layer 2 as logically defined by AVAILABLE.METHODS.

[9]By default, Theo attempts each method in sequence until a successful method is found. Alternatively, Theo can be directed to attempt all the specified methods for slot S and to collect the union of their values, by asserting (S TOGET) = TOGET.SET.

[10]Recall that the SPECIALIZED.DEFINITIONS subslot is inferred via explanation-based learning.

[11]Note here that we view any slot value as an entity address (even numbers), regardless of whether that entity is presently represented by an explicit datastructure with explicit slots.

[12]The function TH is described in detail in Appendix II.

(which may themselves be inferred by Theo) for other slots. Higher level specifications are typically more open to examination by the system. For instance, Theo is able to apply its explanation-based learning mechanism only to slots which use the default value for TOGET (i.e., slots whose inference methods are specified at levels 2 or 3).

4. Representation

This section describes in greater detail the representation conventions of Theo alluded to in the above overview.

Table 4-1: Beliefs about the Husband Entity

```
(husband *novalue*
        (generalizations (spouse))
        (domain female)
        (range male)
        (definitions
                ((onewhich male
                        (equal (th wife *re*) *de*))))
        (specialized.definitions
                ((onewhich male
                        (equal *re* (th father value child *de*)))))
        (default.value nil)
        (plural husbands)
        (singular nil)
        (multivalued? nil)
        (transitive? nil)
        (reflexive? nil)
        (inverse wife)
        (toget toget.first)
        (available.methods (inherits drop.context defines
                            use.inverse default.value))
        (ordered.methods (defines use.inverse inherit
                          drop.context default.value))
        (whentocache always)
        (whentogeneralize always)
        (explanation *novalue*)
        (explanation-data *novalue*)
        (explanation-method *novalue*)
        (dependents *novalue*))
```

Predefined slots of slots. A number of slots for describing slots are defined in the initial Theo knowledge base. These slot slots describe various slot properties such as DEFINITIONS, DOMAIN, RANGE, INVERSE, PLURAL, TRANSITIVE?, and MULTIVALUED?. Other slot slots describe how to infer values of instances of the slot, such as TOGET, METHODS, and AVAILABLE.METHODS, as well as other control information such as WHENTOCACHE, WHENTOEBG. In addition, the slot slots EXPLANATION, DEPENDENTS, EXPLANATION-METHOD, and EXPLANATION-DATA describe information about the interdependencies among slot values. Table 4-1 illustrates some of the system-defined slot slots, in the context of describing the HUSBAND slot. When defining a new slot the user may omit assertions about the values of any or all of these subslots. Theo can operate successfully in the

absence of any of this information, since it will infer default values for subslots that are not explicitly given in the knowledge base. For example, if the user asserts (DENSITY DOMAIN)=PHYSOBJ, then Theo will use this information to constrain its method for inheriting values of instances of the DENSITY slot, so that it searches only among specializations of PHYSOBJ. On the other hand, if the user fails to assert an explicit value for (DENSITY DOMAIN), Theo will infer its value based on the (DOMAIN DEFAULT.VALUE)=ROOT, and will therefore search more widely when attempting to inherit a value for (CUBE1 DENSITY).

Virtual and explicit entities, slots and values. As described above, expressions in Theo can refer to slot values that are not explicitly described by the current datastructures in memory, but which must be inferred on demand. The same is true of entities and slots, so that if Theo is asked to infer the value of some new slot of some new entity, it will construct the entity and the slot, then infer and cache appropriate default methods for inferring the slot value. Note this relieves the user of responsibility for creating entities that define slots about which he has nothing unusual to assert. It also implies that Theo has default methods capable of inferring all essential slot values in the system.

Slots with no value. Even slots that are explicitly described may have no known value. We distinguish between two situations in which no explicit slot value is recorded. If the value is not known and Theo has not yet attempted to infer a value, then the token *NOVALUE* is stored in the value field of the slot. If Theo attempts and fails to infer a slot value, then it stores the token *INFERRED.NOVALUE*, along with an explanation of the methods and slots which justify the absence of a value. By storing *INFERRED.NOVALUE*, Theo avoids subsequent futile searches for values of slots which it cannot infer. Because Theo records an EXPLANATION for each *INFERRED.NOVALUE*, its dependency maintenance mechanism automatically replaces an *INFERRED.NOVALUE* by *NOVALUE* if any change is made which could influence its ability to derive a value for the slot. In practice, we find that a great deal of wasted search effort is eliminated by this scheme of caching *INFERRED.NOVALUE*.

Absolute and relative entity addresses. As noted above, the address of a slot entity is the list of entity names traversed in reaching the slot entity from the top level entity in which it appears (e.g., (CUBE1 WEIGHT DEFINITIONS) is the address of the DEFINITIONS slot of the WEIGHT slot of CUBE1). In addition to such entity addresses (which we will sometimes call *absolute addresses*), Theo also supports *relative addresses*. A relative address is a pair formed by an absolute address and a path specifying how to reach a second entity from the entity addressed by the absolute address. Such relative addresses are convenient when specifying slot DEFINITIONS which refer to addresses of other slots relative to the slot whose value is to be inferred. Appendix II describes Theo's relative address mechanism in some detail.

5. Learning

Three learning mechanisms are presently available in Theo:

- Caching inferred slot values.
- Formulating macro-methods for inferring slot values, based on examining explanations of successful inferences.
- Ordering the methods for inferring slot values, based on inductive inference from statistics of past applications.

In this section, we summarize each of these mechanisms and present experimental data illustrating their individual and combined impact on Theo's performance. [Mitchell, et al 88] provides a more detailed discussion of these learning methods.

5.1. Caching

Caching is the simplest form of learning, in which inferred slot values are stored so that they can be quickly retrieved on subsequent demand. Thus, caching provides a kind of rote learning, in which data is stored without any attempt to generalize it.

Figure 5-1 illustrates the impact of caching in Theo. Note the decreasing cost of a typical slot access in a simple knowledge base, as a result of caching, with Theo's other learning mechanisms disabled. The particular knowledge base defines a set of 12 people (e.g., MEGHAN, SHANNON), a set of 25 slots corresponding to family relations (e.g., SISTERS, UNCLES, DAUGHTERS) and an initial set of beliefs (e.g., (MEGHAN SISTERS)=(SHANNON). Initially, only 17 of the 300 describable beliefs were explicitly given. In this experiment, each of the 300 possible slots were queried, and the cost in seconds of obtaining an answer was measured. Figure 5-1 presents the cpu time cost of each of these 300 slot accesses, with the first slot access at the leftmost point and subsequent slot accesses progressively to the right.

Notice that the cost of the first few dozen slot accesses is quite high, with the cost of subsequent slot accesses dropping dramatically and quickly reaching a plateau. This is largely due to the high initial cost of inferring meta-level beliefs such as (MEGHAN SISTERS AVAILABLE.METHODS), (SISTERS AVAILABLE.METHODS), and (SISTERS DOMAIN). It is also partly due to the need to infer additional ground level beliefs (e.g., SISTERS is defined in terms of PARENTS). Once such information has been inferred and cached during the first few dozen slot accesses, it is directly available when needed to support subsequent slot queries. Thus the high cost of relying on explicit meta-knowledge (e.g., about (SISTERS DOMAIN)) is quickly reduced by caching since most instances of a slot (e.g., instances of the SISTERS slot) make use of precisely the same meta-information.

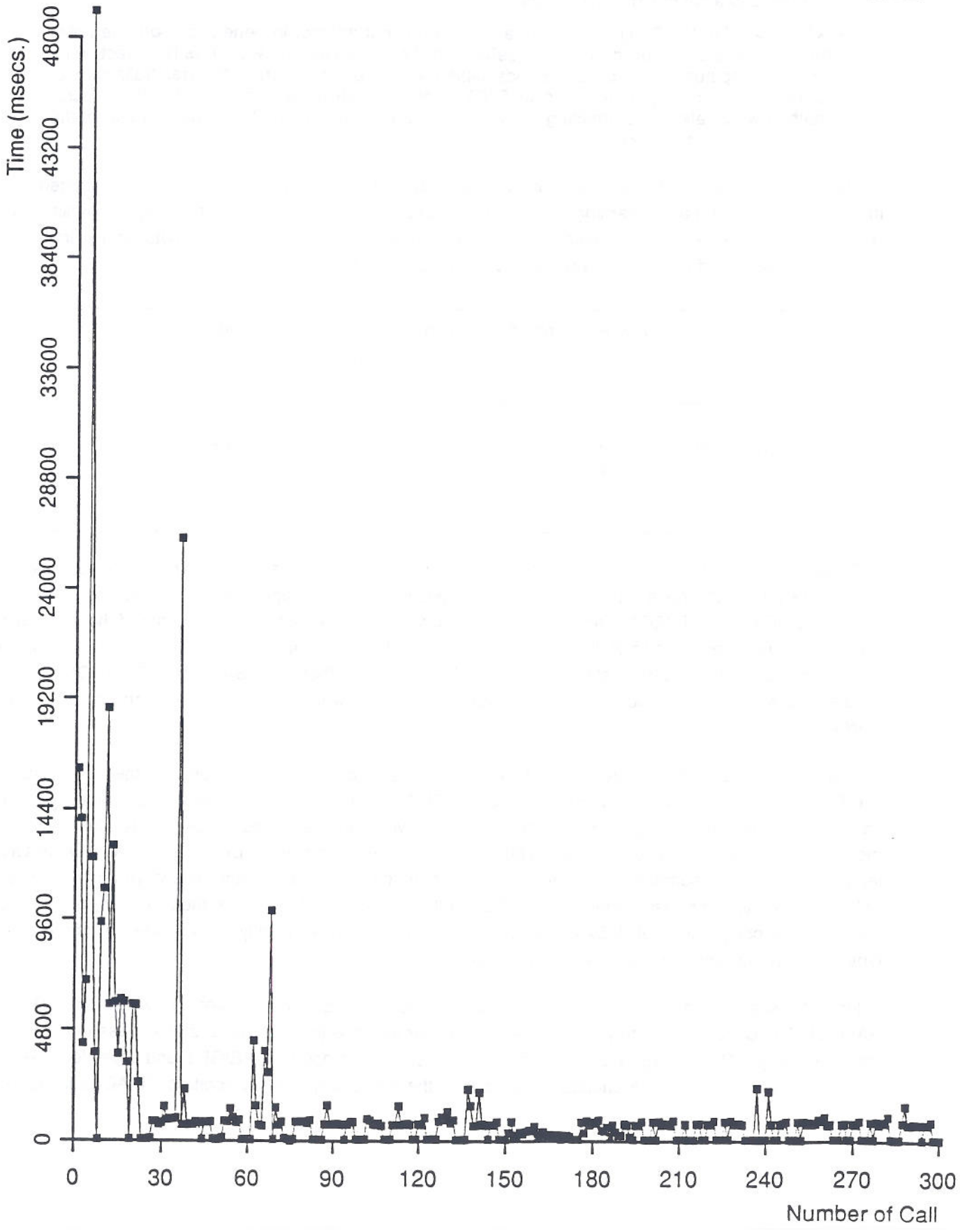
5.2. Explanation-Based Learning of Macro-Methods

As discussed in Section 2, Theo forms macro-methods by examining explanations of previously successful slot inferences. The program which achieves this within Theo is called TMAC. In particular, after Theo infers the value of some slot, S, TMAC can form a macro-method by composing the successful inference steps, computing the generalization, S', of S to which this macro-method can legally be applied, and then storing this macro-method in the SPECIALIZED.DEFINITIONS subslot of S'. Theo will then be able to inherit this macro-method when attempting to infer values for other slots that are specializations of S'.

In general, several possible generalizations may be formed from the explanation associated with a slot value. Consider, for example, the explanation of table 2-3 for the belief (CUBE1 WEIGHT)=5000. From this example, one might apply explanation based learning [Mitchell, et al. 86, DeJong and Mooney 86] to infer sufficient conditions for any of the following target concepts:

- {?e | (?e WEIGHT)=5000}. By determining the general set of entities for which the explanation carries through, it would be possible to determine whether the inferred value for (CUBE1 WEIGHT) can be stored in some more general location in the inheritance hierarchy.

Figure 5-1: Impact of Caching on Cost of Slot Accesses



- {?v | (BOX1 WEIGHT)=?v}. One might alternatively generalize by determining the class of values for which the explanation still applies to the slot instance. In fact, this makes little sense for single-valued slots such as WEIGHT. However, for multi-valued slots such as HEAVIER.OBJECTS, such a generalization could result in generalizing from one legitimate slot value to a set of legal slot values.
- {<?e ?v> | (?e WEIGHT)=?v}. Alternatively, one might attempt to generalize both the entity and the value of the belief, to find a general relation between the two. This is, in fact, what TMAC computes. Its macro-methods define a relation between some generalization of CUBE1 and some generalization of 5000. This relation is defined in terms of a macro-method which allows determining the value ?v from a given entity ?e. Table 2-4 shows the macro-method for this example.

Thus, TMAC accomplishes one type of explanation-based learning, and represents an initial attempt to integrate explanation-based learning with Theo's representation, inference, and indexing mechanisms, as well as its other learning mechanisms. At present, TMAC is applicable only to slots whose inference methods are specified in terms of layers 2 and 3 (Cf. table 3-1).

Table 5-1: Impact of Forming Macro-Methods in Theo

		Time in Seconds	
	Slot Query	TMAC plus Caching	Caching Only
1.	(box1 safe.to.stack.on)	92	118
2.	(box2 safe.to.stack.on)	1.5	3.1

Table 5-1 presents the results of applying TMAC. The data presented here is derived from a knowledge base defining a number of physical objects and their properties (slots) such as WEIGHT, VOLUME, HEAVIER.OBJECTS, and SAFE.TO.STACK.ON. These slots are a superset of those given in Table 2-1, and their definitions form a domain theory similar to that given in [Mitchell, et al. 86] for characterizing when it is safe to stack one physical object on another. The SAFE.TO.STACK.ON slot, for instance, is a slot whose value is a list of physical objects on which the domain element can be safely stacked.

Line 1 of Table 5-1 shows the time in seconds to infer the value of the slot (BOX1 SAFE.TO.STACK.ON), with and without invoking TMAC, but allowing caching of slot values in both cases. Note that even though TMAC carries a certain overhead cost, Theo is able to infer the slot value more quickly when TMAC is included. This is because the inference of (BOX1 SAFE.TO.STACK.ON) leads to accessing a number of additional slots (such as the WEIGHTs of other known physical objects). TMAC is able to infer macro-methods for these other slots, and Theo uses these macro-methods to complete the computation of (BOX1 SAFE.TO.STACK.ON) more efficiently. Thus, line 1 illustrates the type of within-trial learning that TMAC can produce.

Line 2 shows the results of a second, subsequent query to infer a value for (BOX2 SAFE.TO.STACK.ON). As the table shows, the cost for inferring this slot value is much less than for the first slot query. This is largely due to the effects of caching, since the WEIGHT and other features of various physical objects were cached as a result of the first slot query. In addition, TMAC reduces the

cost by a factor of two over the reduction due to caching alone.

While the results in Table 5-1 are typical of those obtained for caching and TMAC, the results depend, of course, on the details of the structure of the knowledge base, the patterns of slot queries, and the frequency with which old slot values are replaced by new values. For example, we find the importance of TMAC relative to caching is minimized when the knowledge base is static (i.e., no old slot values are changed) as in the current example, and maximized when slot values are frequently updated (since this causes dependent values to be removed and macro-methods are very effective for recomputing the new slot values).

5.3. Learning to Order Methods

Since Theo usually has several methods available for inferring the value of any given slot, it is useful for Theo to attempt these methods in some appropriate order. SE is an inductive learning system which can order the AVAILABLE.METHODS for any slot in Theo.[13] SE accepts as input a slot address and the corresponding list of AVAILABLE.METHODS, and produces as output a list of methods ordered in the sequence in which they should be attempted. SE may also be viewed as learning an evaluation function from statistical sampling. As such, it is closely related to the ideas developed independently by [Abramson and Korf 87].

To produce such method orderings, SE utilizes a random sample of executed Theo computations, as well as a predicate language for defining clusters of these example computations. Currently, SE clusters computations by the problem class to which they belong and the method that was used to solve the problem. Each <problem class, method> pair corresponds to a predicate in the predicate language. For example, (WEIGHT INHERIT) corresponds to the predicate satisfied by all executed instances of (WEIGHT INHERIT) such as (BOX1 WEIGHT INHERIT) etc.

By a Theo computation we mean the application of a method (e.g. INHERIT) to infer the value of a specific slot. For each such computation SE considers the time it required and whether it succeeded or failed. Based on this input, SE produces predictions of the probability of success and expected cost for subsequent Theo computations, then uses these predictions to order the AVAILABLE.METHODS of slots. More precisely, given a query from Theo consisting of a slot address and its associated list of AVAILABLE.METHODS, SE sorts the given methods based on a simple function of their probabilities of success and failure, and their expected costs.

The function used by SE to sort the available methods is: cost of success + (probability of failure/probability of success)* cost of failure. [Etzioni 88] presents a general statistical model which has been used in [Etzioni 87] to analyze SE and to prove that SE's method orderings minimize the expected inference cost for Theo, if its cost data is correct. [Etzioni 87] proves that if the estimates used are good, then the ordering will be close to optimal. However, the model used for these proofs makes several assumptions that may not be realistic for Theo. In particular, the model assumes a fixed or slowly changing distribution of computation costs which does not accurately model Theo when caching of slot values is active. Nevertheless, our empirical investigations of SE (see below) indicate that SE is able to significantly improve Theo's computational efficiency.

---

[13]We can only give a brief high level sketch of SE in this context. See [Etzioni 87] and [Etzioni 88] for a complete description.

The crux of SE's problem is generating accurate predictions for the probability of success and cost of each method. To generate its predictions SE groups the sample computations into clusters. A predicate is associated with each cluster, and a computation belongs to the cluster if and only if it matches the predicate. The mean cost of sample computations in the cluster is used as the cost prediction for new computations which fall in the cluster, and the proportion of successful sample computations in the cluster is used as the probability of success prediction for new computations which belong to the cluster. Thus, the cluster's predicate implicitly defines a population of computations. A sample from that population is used to form predictions about the population.

SE attempts to find clusters which are homogeneous with respect to computational cost. That is, the cost of computations in the cluster is more or less the same. For such a cluster, the sample mean is likely to be a good prediction of the cost of new computations which match the cluster's predicate. The quality of SE's predictions depends on its ability to find homogeneous clusters, because the mean is a bad predictor for a nonhomogeneous population.

To summarize, SE is called with a slot address (e.g., (CUBE1 WEIGHT)) and an unordered list of possible methods (e.g., (INHERITS DROP.CONTEXT |WEIGHT.DEFINITIONS.1| USE.INVERSE, DEFAULT.VALUE)). It finds a predicate to match each incipient computation (<method slot> pair). The mean costs, success rate, and failure rate of the sample that matches this predicate are used to obtain predictions for the new computation. Once all such predictions have been made, the method list is sorted and returned to Theo.

Experimental results showing the impact of SE's learning are presented in the following section. Our preliminary experiments indicate that SE is able to significantly speed up Theo computations. However, SE's performance deteriorates when Theo is caching, since SE's data on expected costs becomes outdated over time. We are considering updating schemes for SE's estimates which weight recent data more heavily than older data in order to ameliorate this problem.

5.4. Combined Results of Learning

Table 5-2: Combinations of Learning Mechanisms in Theo

Times in seconds to complete a set of slot accesses

TMAC off

	SE off	SE on
Caching off	1132	641
Caching on	781	526

TMAC on

	SE off	SE on
Caching off	933	425
Caching on	702	389

Table 5-2 summarizes the results of various combinations of Theo's three learning mechanisms, for the task of computing a set of slot inferences involving WIDTH, AREA, and LENGTH of various objects. Note

that the longest time is required when no learning is enabled, and that any one of the learning mechanisms produces a noticeable improvement. Furthermore, combinations of the learning mechanisms produce stronger performance than individual mechansisms, and the strongest improvement is obtained when all three learning mechanisms are enabled.

Although these results are typical of Theo's behavior, the exact payoff of each learning mechanism depends in general on features such as the particular patterns of slot queries and updates, which slots possess inferable values, and the level of independence among slot values. In fact, it is possible to construct worst-case situations in which TMAC and SE lead to deteriorated rather than improved performance in the presence of caching. As a simple example, if one were to repeatedly query a single slot, then caching would provide the optimal learning mechanism and the overhead cost of TMAC and SE would have no offsetting benefits.

We have just begun to explore learning in Theo. Some of the observations and issues that we have identified for further exploration include:

- Each learning mechanism changes the computational environment in a way that impacts the others. The most important example of this involves the impact of caching on SE. Caching slot values changes various computational costs so that SE's estimates of costs for applying various methods can become out of date, and, thus, so can its learned method orderings. One way to deal with this issue of environmental drift might be to weight most heavily the most recent of SE's training data.

- Macro-methods can cause Theo an unnecessary amount of work by causing it to overlook available slot values. For example, when Theo forms a macro-method for HEAVIER.OBJECTS that is based on accessing values of VOLUME and DENSITY of other objects, this macro-method will fail to take advantage of subsequent situations in which Theo holds explicit beliefs about the WEIGHT of various objects.

- Theo should learn about the relative *reliabilities* of its methods. While SE learns to predict the expected costs and probabilities of success of Theo's methods, it should also learn about the reliabilities of these methods. The Default.Value method, for instance, may be highly likely to succeed when invoked, though the value it returns may be less reliable than that obtained via some other method. The control knowledge acquired by SE should take this factor into account. In general, we believe that intelligent systems cannot be constructed solely from infallible inference methods. Taking reliability into account in control knowledge, and recovering from unreliable inferences are high-priority issues on our research agenda.

- Theo should be able to use explanation-based learning as well as SE's inductive mechanisms to order available methods. Theo does not presently utilize explanation-based learning to order its slot methods (it is used only to produce macromethods). In order to achieve this, it may be useful to devise a representation of control knowledge different from the simple ordered list of methods produced by SE. Section 6 discusses this issue further in the context of comparing Theo with the Soar architecture.

- It is generally unclear which of the many possible generalizations that may be derived from a given explanation should be. In deriving a generalization from an explanation, TMAC makes a number of implicit choices that impact the generality/operationality tradeoff of the resulting learned macro-method. Consider, for example, the situation in which the original explanation refers to the default value of a particular slot (e.g., to (WEIGHT DEFAULT.VALUE)=5). When TMAC produces the resulting macro-method, should this method refer explicitly to this slot address, or simply replace the reference by the slot's value? The former leads to a more general, though slightly less operational macro-method (since it requires accessing the slot value at evaluation time). This issue is one of how thoroughly to partially evaluate [Kahn 84] the most general warranted macro-method in order to make it sufficiently operational.

- Theo should be able to decide when it is worth deliberating and when it is worth learning. In general, in acquiring control knowledge, how much deliberation by Theo is warranted in making control decisions? Is it best to make inexpensive unreliable guesses or expensive reliable ones? We suspect that there is no universal answer to this question, that the answer depends on the specific context, and that Theo should be made to reflect on this problem and learn slot-specific answers to the question of how thoroughly to deliberate and learn.

6. Perspective

From one perspective, Theo is a system for storing, inferring, maintaining, and retrieving beliefs, each of the form (<entity> <slot>)=<value>. Some beliefs are about entities from a particular domain (e.g., (CUBE1 WEIGHT)=5000). For each such belief, Theo may hold many more meta-beliefs about the belief itself, and about the problem of inferring the belief (e.g., (CUBE1 WEIGHT AVAILABLE.METHODS), (CUBE1 WEIGHT DEFAULT.VALUE), (CUBE1 WEIGHT EXPLANATION)). Similarly, it can hold meta-beliefs about *these* beliefs and problems as well (e.g., (CUBE1 WEIGHT AVAILABLE.METHODS DEFAULT.VALUE), (CUBE1 WEIGHT AVAILABLE.METHODS DEFAULT.VALUE EXPLANATION)). Note that even though Theo's potential beliefs cover an indefinite number of meta-levels with an increasing number of potential beliefs at each level, there is no necessary difficulty with infinite regress or with interminable meta-level reasoning. Theo need not necessarily consider the infinite number of meta-level problems which it can in principle pose, or the infinite number of meta-level beliefs which it can in principle hold.

Under this view, problem solving is the process of inferring the <value> field of a belief, given its <entity> and <slot>. Theo solves such problems by applying methods it believes to be appropriate to that problem, and by controlling the problem solving search based on yet other beliefs about the problem. Because Theo's ground and meta beliefs are described in a single notation, it can apply the same inference methods to both. This perspective raises questions about the distinction between learning and problem solving. Both are simply means for producing new beliefs about some entity.

A related perspective on Theo is that it is an infinitely large virtual datastructure which is incrementally made explicit on demand. This virtual datastructure corresponds to the set of all beliefs which Theo can in principle hold, and all problems which Theo could ever in principle pose. Furthermore, each such belief and problem has a name (i.e., its slot address) which provides an efficient means for indexing and retrieving the belief or problem in this datastructure. If this (infinite) virtual datastructure could be made completely explicit, then Theo could solve all subsequent problems it encountered by simply looking up their solutions. Of course this is not possible, but it does provide a useful perspective for understanding the significance of simple caching of inferred beliefs in a uniform representation with an efficient associated indexing mechanism. In fact, Theo can never make this datastructure completely explicit, and it simply makes those parts explicit that correspond to problems it encounters. Furthermore, its learning mechanisms seek to generalize its experience so that information is stored with problem classes rather than with the individual problem instances that constitute its experience.

This perspective raises questions about properties of this virtual datastructure as it becomes increasingly explicit. Is it possible to characterize Theo's level of competence at solving some class of problems as this datastructure becomes increasingly explicit? Can one show that Theo's indexing scheme based on absolute and relative slot addresses allows an indefinite increase in explicitness of this datastructure with a bounded increase in the cost of accessing knowledge needed to solve new problem

instances? Does Theo's virtual datastructure include the deductive closure of its explicit beliefs? We are currently considering such questions about Theo's representation, inference, and indexing mechanisms.

6.1. Comparison with Soar and RLL

The design of Theo has been influenced by several earlier efforts to develop uniform architectures for self-improving systems, especially by Soar [Laird, et al. 87] and RLL [Greiner and Lenat 80]. Here we consider the relationship between Theo and these two systems.

Soar [Laird, et al. 87] is an architecture intended as a model for human cognition, and is organized around the notion of problem spaces. Problem solving corresponds to search in some problem space, and the decisions which must be made to guide this search (e.g., What are the legal moves? Which of these should be preferred as the next search step?) are made by applying a set of production rules to infer the information directly. When productions are not available to make such decisions, Soar recovers from this impasse by formulating the decision as a problem to be solved by search through some other problem space. Once this decision problem is solved, Soar compiles, or "chunks" the answer into a production rule which can provide the answer directly for similar subsequent problems. Thus, chunking provides the mechanism in Soar for incrementally compiling information which is in the inferential closure of Soar's knowledge, but which is not initially explicit in its productions. Soar has a *complete* set of impasses, in the sense that *every* decision involved in conducting search in a problem space can be formulated as a problem to be solved in another problem space, and Soar possesses weak methods to solve all such problems.

RLL [Greiner and Lenat 80] is a frame-based architecture which has been used as the basis for implementing the Eurisko program [Lenat 83]. Eurisko extends the initial set of frames and slots that describe some domain (e.g., mathematics, VLSI structures) in order to explore the space of possibly useful new concepts in that domain. The important points here are RLL's ability to explicitly describe its own structure and its ability to support automatic generation of new frames and slots which constitute the vocabulary of its domain-specific representations.

Theo represents an attempt to incorporate impasse-driven computation similar to that found in Soar, with a self-describing frame-based representation similar to that found in RLL, along with explicit representation of explanations for beliefs, the ability to describe and index beliefs about *every* problem and belief, and a variety of inference and learning mechanisms.

Consider the correspondence between Soar and Theo:

- **Uniform problem representation.** Both Soar and Theo are organized around a uniform representation of all problems. Soar is organized around problem spaces, and Theo around problems described by pairs of the form (<entity> <slot>).
- **Problem solving knowledge indexed to problem description.** Both systems index problem solving knowledge based on the problem description. In Soar, problem solving knowledge is stored in productions whose conditions test the problem space description. In Theo, problem solving knowledge resides in the subslots of the (<entity> <slot>). Note that in Theo, no condition matching is needed to index cached problem solving knowledge.
- **Use of impasse-driven computation.** Both Soar and Theo use impasse-driven computation to infer necessary problem solving knowledge when this knowledge is not directly retrievable. In Soar, such impasses lead to solving problems in another problem space. In Theo, impasses correspond to queries of slots whose values are unknown, and

result in the subtask of inferring the corresponding slot value. In both cases, an impasse in solving one problem results in formulating and solving a second problem described in the same representation.

- **Caching results of impasses.** Both systems cache the problem solutions resulting from such impasses. In Soar, solutions to impasses are stored in productions whose condition elements mention only those features of the problem which were used to resolve the impasse. In Theo, solutions to impasses are simply the inferred slot values, and are stored directly in the subslots of the problem instance. Note that chunking in Soar indexes the result of the impasse in a way that allows it to be retrieved more generally. Caching in Theo does not do this, although Theo's explanation-based learning component (i.e., TMAC) does. Thus, explanation-based learning in Theo plays a role similar to that played by the generalization step in chunking in Soar. See [Rosenbloom and Laird 86] for a discussion of the relationship between chunking in Soar and explanation-based learning.

The above list makes clear several major similarities between Theo and Soar, and the correspondence between problems as (<entity> <slot>) pairs in Theo and problem spaces in Soar. There are also significant differences:

- **Different representations for control knowledge.** In Soar, knowledge to control search is expressed by explicitly asserted preferences (e.g., operator1 is preferred to operator2). In Theo, control knowledge is currently described by a sorted list of methods, but there are no explicitly asserted preferences between individual methods in this list. Thus, Soar's representation of control knowledge is finer grained: its individual preferences allow describing a partial rather that total ordering on the methods, and this partial ordering is often all that need be infered in order to procede with selecting a method. It may be advantageous to elaborate the representation of preferences among methods in Theo in a similar fashion.
- **Difference in uniformity of representation.** Theo's slots avoid the distinction made in Soar between problem spaces and attributes used in data representations (e.g., attributes used to describe problem states). This additional uniformity in Theo's representation allows it to apply the same inference mechanisms to resolving impasses that it applies to inferring the value of some attribute of the problem state. In fact, there is no fixed set of impasses in Theo, since *any* slot whose value is needed but not available can lead to attacking a new subproblem.
- **Different commitments to permanence and explicitness of explanations.** Theo records permanent explicit explanations for its beliefs, and these can be examined by the system. They are also used by Theo to maintain dependencies and to remove beliefs whose explanations cease to hold. Soar uses similar dependencies when constructing its "chunks", but it neither records these dependencies permanently, nor makes them available to the non-architecture portion of the system. Soar thus seems to have no corresponding notion of belief maintenance, and it is unclear how Soar deals with situations in which one of its productions becomes outdated because of changed beliefs that have been compiled into it.
- **Different memory indexing schemes.** Soar's long term memory consists solely of productions, while Theo's consists solely of entities and beliefs about them. These two memory organizations are similar in some sense, since one can typically translate an If-Then production into a belief about some appropriate class of entities. The differences in storage costs and retrieval speeds between these two schemes are poorly understood, however, and these issues are critical ones as we scale to larger knowledge bases.
- **Difference in use of explicit beliefs about problem classes.** Theo explicitly represents beliefs about problem classes, such as beliefs about their domain, range, inverse, generalizations, and specializations. It uses this information to infer solutions to the problems (e.g., Inheritance uses the slot's DOMAIN, Use.Inverse uses the slot's INVERSE, and Slotspecs uses a slot's SPECIALIZATIONS). This focus on problem classes as objects for reasoning is quite different from that in Soar. In Theo, approximately 80 slots, or problem classes, are predefined when the system is loaded, and Theo commonly caches new beliefs

about the DOMAIN, AVAILABLE.METHODS, etc. of its taxonomy of problem classes.

- **Different learning mechanisms.** Soar presently has only one learning mechanism, and Theo three. As noted above, chunking in Soar is used to cover the same kinds of learning tasks that caching and explanation-based learning cover in Theo. Theo's third learning mechanism, the inductive inference method for inferring method orderings, has no inductive counterpart in Soar (whose control information is inferred by chunking). In some sense, Soar is more successful in integrating a single learning mechanism into the architecture (perhaps due to its representation of control information in terms of explicit preferences). However, it is not clear how Soar would accomplish the kind of statistical inference performed by SE.

The above comparison between Soar and Theo is based on viewing both as instances of impasse-based architectures which incrementally make explicit information from their inferential closure. In contrast, the following comparison between RLL and Theo is based on viewing both as self-descriptive, frame-based representation systems.

As noted above, RLL [Greiner and Lenat 80] is a modifiable and highly self-descriptive representation system which has served as the basis for implementing the Eurisko program [Lenat 83]. RLL achieves it's functionality by utilizing a highly uniform representation of frames and slots, and by possessing declarative descriptions of parts of itself. It allows describing each slot (e.g., VOLUME) by a frame whose slots describe beliefs about the slot (e.g., (VOLUME DOMAIN)=PHYSICAL.OBJECTS), including information about how to infer values of the slot. It allows organizing such slot-defining frames into taxonomies so that their properties can be inherited, and allows defining new slots (e.g., FATHER) in terms of existing slots (e.g., SPOUSE, MOTHER). In addition, RLL attempts to provide declarative representations of methods with the goal of making it easy to create new methods by copying and editing the definitions of existing related methods.

Theo builds on many of the ideas in RLL, including the central ideas of explicit entities to describe each item in the system, and slots that specify beliefs about other slots, including information about how to infer their values. As in RLL, each slot in Theo has a corresponding top-level entity (e.g., VOLUME) whose slots define its properties in general. However, Theo also has entities corresponding to *each use of each slot*. This extension results in the important property that Theo can pose problems and hold beliefs about *every specialization and instance* of the VOLUME relation in the system. Furthermore, it can hold beliefs about these beliefs, and so on. Consider, for example, the definitions for various specializations of VOLUME:

- (VOLUME DEFINITIONS)= Integral over the surface of dx dy dz
- (PARALLELPIPED VOLUME DEFINITIONS)= Length x Width x Height
- (CUBE VOLUME DEFINITIONS)= Length x Length x Length

Without the ability to refer to the (CUBE VOLUME) specialization of the VOLUME relation, Theo would not be able to assert its specialized definition. In fact, one could create a new slot name, CUBE-VOLUME, assert that this is a specialization of the slot VOLUME, and store the specialized definition there. However, one quickly encounters an explosion of atomic slot names if this practice is followed for every specialization and instance of each slot. Consider, for example, the explosion resulting from constructing special-case names for each instance of the EXPLANATION slot (e.g., (CUBE1 VOLUME DEFINITIONS EXPLANATION), (CUBE2 VOLUME DEFINITIONS EXPLANATION), ...). This ability to specify beliefs about beliefs and about problem classes is especially important in representing specialized problem solving knowledge about subclasses of problems.

6.2. Conclusion

Theo is still evolving as an architecture. It constitutes the focus of our current efforts to understand the variety of representation, inference, self-reflection, learning, and memory indexing issues that arise in constructing general intelligent agents, and especially the interactions among these issues. For years, the field of Artificial Intelligence has taken a reductionist approach to addressing these issues, studying them separately with the belief that the eventual solutions could be combined later. Our working assumption is that this reductionist research strategy has reached the point of diminishing returns. Our hope is that by constructing unified architectures such as Theo, we will be able to more effectively study these issues in their full context. For example, many instances of problem solving and learning tasks occur within Theo, and for each it is possible to understand the full context in which it occurs, the full set of knowledge and data available to the system in addressing the task, and the costs and benefits of succeeding or failing at the task. Our goal is to use Theo to develop a better understanding of these individual issues in their full context, as well as the interactions among them.

7. Acknowledgements

Many people have contributed ideas over a long period of time in discussions regarding the design of Theo. We thank John Laird, Allen Newell, and Ron van Kempen for useful comments on an earlier draft of this paper. This work has been supported by the National Science Foundation under grant IRI-8740522, and in part by a grant from Digital Equipment Corporation.

I. Inference Methods

Table I-1: Inference Methods that Can Be Specified in AVAILABLE.METHODS

Method	Method Definition
DEFAULT.VALUE	Retrieve a value from the DEFAULT.VALUE subslot of the given slot. Example: (BOX VOLUME) <-- (BOX VOLUME DEFAULT.VALUE)
INHERITS	Inherit values from GENERALIZATIONS of the domain element of the slot. Example: (CUBE1 VOLUME AVAILABLE.METHODS) <-- (CUBE VOLUME AVAILABLE.METHODS) Example: (WEIGHT DOMAIN) <-- (PHYSOBJ.SLOT DOMAIN)
INHERITS.FROM.CONTEXT	Inherit values from a more general context, but *not* from GENERALIZATIONS of the immediate slot name. Example: (BOX VOLUME DEFINITIONS) <-- (PHYSOBJ VOLUME DEFINITIONS) Non-example: (VOLUME DEFINITIONS) <-- (PHYSOBJ.SLOT DEFINITIONS)
DROP.CONTEXT	Retrieve a value from the slot given by the tail of the address. Example: (BOX VOLUME DEFINITIONS) <-- (VOLUME DEFINITIONS)
DEFINES	Infer a value from each of the expressions in the DEFINITIONS or SPECIALIZED.DEFINITIONS subslot.[14] Example: (CUBE1 WEIGHT) <-- (* (CUBE1 VOLUME) (CUBE1 DENSITY)) where (CUBE1 DEFINITIONS) includes (* (TH DENSITY *DE*) (TH VOLUME *DE*))
SLOTSPECS	Retrieve a value from some specialization of the slot. Example: (FRANK CHILDREN) <-- (FRANK DAUGHTERS) where value of (CHILDREN SPECIALIZATIONS) includes DAUGHTERS
USE.INVERSE	Infer a value based on the value of the inverse of this slot. Example: (FRED WIFE) <-- WILMA where (WILMA HUSBAND)=FRED and (WIFE INVERSE)=HUSBAND

Table I-1 lists the inference methods presently defined in Theo.

[14] In fact, the procedure for DEFINES is somewhat less direct than described here. Theo uses the values of the DEFINITIONS and SPECIALIZED.DEFINITIONS subslots to construct Lisp functions which it then applies to the address of the domain element.

II. Relative Addresses

Table II-1: Relative Addressing

```
(mary *novalue*
    (mother sue
        (entity.author john)))

(sue *novalue*
    (entity.author frank)
    (mother ann
        (entity.author harry)))
```

Examples of relative addressing:

```
1A.Relative address: (mary (mother))
   Equivalent absolute address: (mary mother)

2A.Relative address: (mary (mother entity.author))
   Equivalent absolute address: (mary mother entity.author)

3A.Relative address: (mary (mother value entity.author))
   Equivalent absolute address: (sue entity.author)

4A.Relative address: (mary (mother value mother entity.author))
   Equivalent absolute address: (sue mother entity.author)
```

Examples of using TH to refer to slot values based on relative addressing:

```
1B. (th mother 'mary) = sue

2B. (th entity.author mother 'mary) = john

3B. (th entity.author value mother 'mary) = frank

4B. (th entity.author mother value mother 'mary) = harry
```

Relative addresses in Theo provide a convenient means of addressing one entity relative to another. This addressing method is useful in describing general slot definitions which index other slots relative to the slot whose value is to be inferred.

A relative address is a pair formed by an absolute address and a path specifying how to reach a second entity from the entity addressed by the absolute address. The path is a list of slot names and the token "VALUE," which specifies a path of subslots or slot values. The slot names in the path specify subslots of the starting entity which are to be traversed to reach a subentity, and the token "VALUE" refers to the entity named in the value field of the current slot entity. The syntax of absolute and relative addresses may be summarized as follows (where the symbol <x>+ stands for one or more instances of <x>):

<absolute-addr> := <entity-name> | (<entity-name>+)

<relative-addr> := (<absolute-addr> <path>)

<path> := (<path-token>+)

<path-token> := <entity-name> | VALUE

Table II-1 illustrates the use of relative slot addresses. In example 2A, for instance, the relative address consists of the absolute address MARY, plus the path (MOTHER ENTITY.AUTHOR), and thus refers to the entity whose absolute address is (MARY MOTHER ENTITY.AUTHOR). In example 3A the token "VALUE" appears in the path. Thus, the relative address in this case refers to the ENTITY.AUTHOR subslot of the VALUE of the ENTITY.AUTHOR subslot of MARY. This is the entity whose absolute address is (SUE ENTITY.AUTHOR).

One useful function in Theo is the function TH, which accesses slot values via relative addresses. TH takes the relative address of a slot (in a modified syntax) and returns the value of that slot. Examples 1B through 4B in Table II-1 show the use of TH for the same relative addresses as examples 1A through 4A. The arguments to TH are the <path-tokens> of the relative address given in reverse order, followed by an expression which must evaluate to the <absolute-address>. TH is especially useful in describing values of slot DEFINITIONS, since it provides a mechanism for indexing beliefs relative to the slot whose value is to be inferred.

References

[Abramson and Korf 87]
Abramson, B., and Korf, R.E.
A Model of Two-Player Evaluation Functions.
In *Proceedings of the 1987 National Conference on Artificial Intelligence*. AAAI, Morgan-Kaufmann, July, 1987.

[DeJong and Mooney 86]
DeJong, G., and Mooney, R.
Explanation-Based Learning: An Alternative View.
Machine Learning 1(2):145-176, 1986.

[Dietterich and Michalski 83]
Dietterich, T.G., and Michalski, R.S.
A Comparative Review of Selected Methods for Learning from Examples.
Machine Learning: An Artificial Intelligence Approach.
Tioga Press, Palo Alto, CA, 1983, pages 41-82, Chapter 3.

[Etzioni 87]
Etzioni, O.
A Statistical Approach to Learning Control Knowledge.
1987.
Internal Theo Project Working Paper.

[Etzioni 88]
Etzioni, O.
Hypothesis Filtering: A Practical Approach to Reliable Learning.
In *Proceedings of the Fifth International Conference on Machine Learning*. Morgan-Kaufmann, June, 1988.

[Greiner and Lenat 80]
Greiner, R. and Lenat, D.B.
A Representation Language Language.
In *Proceedings of the First National Conference on Artificial Intelligence*. AAAI, Morgan-Kaufmann, August, 1980.

[Kahn 84]
Kahn, K.M.
Partial Evaluation, Programming Methodology, and Artificial Intelligence.
The AI Magazine 5(1):53-57, Spring, 1984.

[Laird, et al. 87]
Laird, J., Newell, A., and Rosenbloom, P.
SOAR: An Architecture for General Intelligence.
Artificial Intelligence 33(1):1-64, September, 1987.

[Lenat 83]
Lenat, D.B.
Eurisko: A Program that Learns New Heuristics and Domain Concepts.
Artificial Intelligence 21(1):61-98, March, 1983.

[Minton 88]
Minton, S.
Quantitative Results Concerning the Utility of Explanation-based Learning.
In *Proceedings of the 1988 National Conference on Artificial Intelligence*. AAAI, Morgan-Kaufmann, August, 1988.

[Minton, et al 87]
Minton, S., Carbonell, J.G., Etzioni, O., Knoblock, C.A., Kuokka, D.R.
Acquiring Effective Search Control Rules: Explanation-Based Learning in the PRODIGY System.
In Langley, P. (editors), *Proceedings of the Fourth International Workshop on Machine Learning*. Morgan-Kaufmann, Irvine, June, 1987.

[Mitchell 83] Mitchell, T.M.
Learning and Problem Solving.
In *Proceedings of the Fifth International Joint Conference on Artificial Intellignece.* IJCAI, Morgan-Kaufmann, August, 1983.

[Mitchell, et al 88] Mitchell, T.M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., and Schlimmer, J.
Theo: Integrating Multiple Learning and Inference Mechanisms .
1988.
Forthcomming CMU CS Technical Report.

[Mitchell, et al. 86] Mitchell, T.M., Keller, R.K., and Kedar-Cabelli, S.
Explanation-Based Generalization: A Unifying View.
Machine Learning 1(1), 1986.

[Rosenbloom and Laird 86]
Rosenbloom, P.S, and Laird, J.E.
Mapping Explanation-Based Generalization Onto Soar.
In *Proceedings of the 1986 National Conference on Artificial Intelligence.* AAAI, Morgan-Kaufmann, July, 1986.

[Schlimmer 87] Schlimmer, J.
General Search in Theo.
November, 1987.
Internal Theo Project Working Paper.