DRAFT

Towards a General Knowledge Representation Language

John Ramish
Advised by Prof. Tom Mitchell

Abstract

In the history of AI, there have been many different knowledge representation languages for different purposes: production systems for fine-grain control, logics for propositions, frames for categories, programming languages for abstract procedures, Bayes nets for probabilistic reasoning, and others. Each language is good at representing certain types of knowledge (e.g. uncertainty, groups, patterns), but poor at representing many others. A complete AI agent, however, must be able to represent all major types of knowledge. This thesis has begun to address this problem of designing a general knowledge representation language.

The first part of this thesis organized the research program. We first present insights on knowledge representation from many disciplines (cognitive psychology, linguistics, common sense reasoning, etc.). Many such ideas clash with the current paradigm: human knowledge is context-specific, four-year olds can reason in second-order logic, and first-order logic is a small subset of English. We then identify the major types of knowledge. Next we present benchmark problems for measuring the capabilities of proposed languages. Finally, we identify the strengths and weaknesses of the major existing knowledge representation languages.

For the second part of this thesis, we developed a new knowledge representation language (or, more aptly, cognitive system) that aims to integrate as many types of knowledge as possible. The system draws ideas from many existing languages. It aspires to the declarative expressiveness of logics, the conceptual clarity of frames, the fine-grain control of production systems, and the abstract procedural organization of programming languages. We present a detailed specification of the system, analyze its capabilities, and demonstrate it many knowledge representation problems.

Contents

Introduction

Knowledge representation aims to provide formal syntax and semantics to concepts, facts, rules, procedures, and other knowledge. For example, the English statement "All apples are fruits" can be formally represented in logic as "forAll x apple(x) => fruit(x)". Similarly, addition can be formally represented as a procedure. Knowledge representation provides an approach to the ultimate goal of AI through the following steps:

1) Design internal representation language
2) Build in primitive concepts
3) Construct interface: natural language processing
   Optionally: perception, robotics
4) Educate

Variants of this approach have been suggested by many researchers, from Turing to Minsky (2004) to Lenat. Lenat's CYC project aimed ultimately to implement nearly this

exact approach, albeit using second-order logic instead of performing step (1); we claim that this was a critical omission, and that the resulting system is crippled by its inability to represent many types of knowledge (most notably procedural knowledge).

One point of contention in the above approach to AI is the role of perception and robotics. For the purposes of this project, we have taken the stance that interesting and useful disembodied intelligence can be acquired solely through natural language, and have focused on this approach.

At any rate, over the course of the past half century, the field of knowledge representation (classically, knowledge representation and reasoning – we prefer to regard reasoning as a particular variety of procedural knowledge) has produced many representation languages, often for purposes less ambitious than the ultimate goal of general AI. There are now many separate representations, from first-order logic to production systems to frames, for many different kinds of knowledge, from propositions to rules to categories. A complete AI agent, however, must be able to represent all of these within a unified framework.

Moreover, as researchers have turned attention to common sense reasoning (e.g. (Minsky, 2004)), they have found a wealth of ideas that cannot be adequately represented in any existing framework. A complete agent must have a more powerful language that can represent these ideas.

Brachman and Levesque (p. 328) have suggested that the search for more powerful representations is futile, since the more expressive a language is, the less tractable reasoning within it is. Although this is undoubtedly true for reasoning in general, it is not true for the types of reasoning that occur in practice. Humans, for instance, appear to employ powerful representations tamed by correspondingly powerful procedural biases towards problems that occur in practice. Finding ways to incorporate these procedural biases is one of the major challenges in this field.

Minsky (2004), on the other hand, has argued that we shouldn't look for a single general purpose representation, but rather for ways to conglomerate many representations that are good for different types of problems. Although it is certainly desirable to have many ways of thinking about problems, we think these should all fit into a unified architecture. In particular, if one representation is capable of representing uncertainty and another is capable of representing the beliefs of other agents, then a conglomerate representation may be able to represent uncertainty and beliefs separately, but won't be able to handle uncertain reasoning about beliefs. For this, a unified framework is needed.

This research project addressed this problem in two parts: analysis of the state of knowledge representation, and design of a new knowledge representation language.

Part I: Analysis of the State of Knowledge Representation

The first part of this project sought to organize and analyze the problem by answering these questions:

-What do other disciplines such as cognitive development and linguistics suggest about the form of a complete knowledge representation language?

-What are the major types of knowledge a complete agent should be able to represent?
-How can we test the capabilities of a proposed knowledge representation language?
-What ideas can existing languages represent easily (in practice, not principle)?
-What ideas in different frameworks are in need of integration in a unified framework?
-What ideas cannot be adequately represented in any existing formal language?

Other Perspectives and Insights

Cognitive Psychology

Research in cognitive psychology depicts the human mind as the near antithesis of a domain-independent theorem prover: human intelligence appears to be context-dependent and scruffy. Untrained humans are poor at pure logical and probabilistic inference, but display robustness on real world inferences and problems (Anderson, p. 351). Human intelligence appears not to be a general capacity, but rather many context-specific capacities covering the space of problems in a broad range of domains (Anderson, p. 306). Reasoning is not the only important thought process; many other procedures (e.g. addition, how to drive) and rules of behavior (how to go to a restaurant) cannot be considered reasoning in any sense. Similarly, nominal kinds (or categories) of objects like "circle" and "uncle," which are defined by human convention and are the emphasis in knowledge representation, are not the only types of kinds. There are also natural kinds (animals, plants, etc.) and artifacts (cups, tables, etc.) (Flavell, p. 111) which are much harder to formally represent. Moreover, pattern recognition is not only for perception, as some proponents of symbolic approaches might like to believe, but also for abstract thought. According to work done by Simon (Anderson, p. 301), for instance, the way humans play chess involves learning to recognize 50,000 strategic situations and to produce appropriate responses for each. Finally, we note that humans routinely solve ill-defined problems: it is possible and often useful to produce effective behavior without rigorously justifying it. These results suggest a much scruffier, pragmatic, context-dependent approach to knowledge representation than is currently in style in AI.

Cognitive Development

The field of cognitive development shows just how stupid and how smart children can be. Children are not born knowing many "obvious" facts such as the fact that the amount of liquid is conserved when it is poured from one short, fat beaker into another tall, thin one (Flavell, p. 140). At the same time, four-year-olds can reason in second order logic (quantifying over properties) since they understand that people inherit most physical properties (e.g. tall, fat, crooked nose) from their parents (Flavell, p. 121). It is also remarkable to observe the pervasive role of imagination, fantasy, and creativity in children, who pretend to be firemen, believe in Santa Claus, and read stories about talking animals. The earliness and universality of these abilities suggest that they are at the core of intelligence.

Development is also fundamental to human knowledge representation. Children undergo several qualitative changes in their representational capabilities (Flavell, p. 4), and the representation languages of adults continue to evolve. The human internal representation language is fluid, evolving, and dynamic, constantly assimilating knowledge and accommodating it (Flavell, p. 5). Siegler's overlapping waves model (Flavell, p. 15) claims humans maintain many different context-specific strategies which compete, flourish, and decline over time as some prove more successful than others. The practical representational capabilities evolve with the internal representation languages, as humans chunk ideas, change primitives, and learn new fundamental concepts and ideas. This fluidity is not only useful, but also necessary for effective knowledge representation.

Common sense reasoning

Common sense ideas, which constitute the core of human knowledge, are approximate and tightly integrated across many domains. Minsky (1986, p. 22) and Lenat emphasize the difficulty of identifying common sense ideas: many are things we don't even realize we know.

The field of common sense also shows that the fact "All men are mortal" is far from typical. The fact "Leaves fall off trees" is more typical. But not every leaf falls off every tree, and not every kind of tree loses any leaves at all. Moreover, the process generally occurs in the fall, but the fall of an individual leaf is unpredictable. It's a social norm to remove the fallen leaves from your lawn, but people don't care about them in the woods. Finally, children can earn money by raking and bagging leaves for people. Despite the apparent hidden complexity of all this, children understand these things. Indeed, Minsky (2004) argues that just to understand a simple children's story requires many types of knowledge – most of the types of knowledge identified in a later section.

Linguistics (universal language concepts)

Perhaps the most striking fact about English from the knowledge representation standpoint is how little of it can be properly represented in first order logic or any other existing representation. Consider representing the mental concepts of "conceive," "consider," "guess," and "understand" for instance. Harrison (1996) has identified a set of universal concepts in human languages. Among them are such representational challenges as "can," "should," "kiss," "kick," "across," and "sporadic." There are many common concepts that current representation languages don't even attempt.

Language also reveals the fuzzy, interconnected nature of human concepts. As Minsky claims (2004), it is not a coincidence that "transfer," "transport," and "transmit" share the root "trans." Lakoff (1994) has identified more than seven different ways in which humans conceive of "ideas," from food ("half-baked ideas") to locations ("we're getting off track") to objects ("I have an idea"). The use of metaphor and analogy seems to be the norm, not the exception. Minsky (2004) has argued that reasoning by analogy is one of our most powerful strategies, and this requires a representation of similarities.

Mathematical problem solving

The field of mathematical problem solving presents an interesting perspective, since mathematics is considered to be the most formal of disciplines and a promising

candidate for automation. The field suggests, however, that human mathematical problem solving is not so different from other types, and is not so similar to automated theorem proving. Mathematical problem solving involves pattern recognition, abstract strategies, and the synthesis of many ideas in an open-ended way. Many mathematical arguments cannot even be written purely in symbols without English. Consider attempting to represent these common strategies from Larson (1983): search for a pattern, draw a figure, choose effective notation, consider extreme cases, and generalize. Working towards representing even one of these strategies in some formal way would be instructive for further progress.

Types of knowledge

Based on the many previously discussed fields and on existing knowledge representation languages, we have identified the major types of knowledge shown below. We have chosen the classical declarative and procedural decomposition since this provides a particularly good way to compare the declaratively strong and procedurally strong representation languages. Declarative knowledge is "knowing that," such as the fact that "All apples are fruits." Procedural knowledge is "knowing how," such as how to add.

It is important to note the distinction between organizing knowledge by type, as we have done it, and by domain, as it has traditionally been done. Previous organizations of knowledge have been into different domains such as naïve physics, folk psychology, and spatial knowledge. For the purposes of designing a language, however, the underlying type of knowledge (e.g. categories, uncertainty, context, or control) is more relevant. Finally, we note that these types of knowledge overlap widely: they are more shades of meaning than distinct categories.

Declarative knowledge

Groups
: abstraction, common characteristics, categories, groups, hierarchies, clusters, universal quantification

Relations
: objects, pointers, functions, relations, properties

Change
: sequences, time, events, change, actions, causality

Uncertainty
: uncertainty, ignorance, vagueness, approximate concepts, partial knowledge, possibility

Higher-order ideas
: higher-order ideas, reification, recursion, nested ideas, compositionality

Meta-knowledge
: meta-knowledge, self-knowledge, reflectivity, reflexivity, episodic memory, declarative knowledge of procedures

Context
: working memory, context, situation, current pattern

Procedural knowledge

Problem solving
: problem solving, search, planning

Reasoning
: reasoning, inference, generalization, inheritance

Control
: control, context-specific heuristics, attention, rules, strategy

Retrieval
: memory, indexing, retrieval

Development
: development, accommodation, extendibility, modifiability, restructuring knowledge, learning representational bias, non-monotonic reasoning, chunking, learning control, memoization, explanation based learning, proceduralization of declarative knowledge

Sequential procedures
: sequential procedures, recursion, arbitrary algorithms

Meta-cognition
: meta-cognition, self-modification, reflectivity, reflexivity, meta-learning, meta-reasoning

Pattern recognition
: pattern recognition, similarity, assimilation, analogies, defaults

Pattern generation
: pattern generation, imagination, creativity, constructive memory

Knowledge representation benchmark problems

We promote a pragmatic approach to knowledge representation, based on concrete demonstrations of representational capabilities on real world problems instead of theoretical proofs. Systematically organized standard benchmark problems for representational capabilities have been conspicuously absent in the field of knowledge representation. Leora Morgenstern maintains a set of commonsense problems online at http://www-formal.stanford.edu/leora/commonsense/. It contains about 25 problems from a small number of sources, and is organized by domain.

We present another effort. Our benchmark problems are for knowledge representation in general (not just commonsense knowledge). There are about 20 pages of problems available online (search for "John Ramish benchmark problems"), drawn from a variety of disciplines including artificial intelligence, cognitive psychology, cognitive development, and mathematical problem solving. The problems range in difficulty from routine problems (e.g. "All birds have skin. All canaries are birds. Infer that canaries have skin.") to open research problems (e.g. "Play chess as Herbert Simon proposes humans do, using 50,000 situation-specific rules. Learn them."). The problems are organized by the types of knowledge identified in the previous section and are doubly linked, so that each knowledge type has an associated list of problems and each problem

has an associated list of knowledge types. This list of problems is woefully incomplete, as any such general list must be. It is meant to provide a coherent picture of the range of knowledge types through representative problems from many disciplines.

In general, there are at least two possible types of benchmark problems for knowledge representation languages: syntactic and semantic problems. Syntactic problems (sentences of English) simply test whether a proposed language claims to have a syntactic expression for a statement in English. Semantic problems (given sentences combined with questions in English) test further whether the proposed language can demonstrate certain types of understanding of the semantics of English statements.

Unfortunately, unlike fields like machine learning, the assessment of performance on benchmark problems is inherently somewhat qualitative instead of quantitative. The reason for this is that the goal is to assess whether a formal representation covers the meaning of an informal statement of English in a general way ("is that really what the English means?"). With enough semantic problems, it should be possible to make evaluation more quantitative. At any rate, qualitative assessments and comparisons are certainly better than none at all.

In fact, benchmark problems in knowledge representation are useful for at least the following purposes:

-They provide a standard for comparisons of languages

-They promote robustness

-They promote goal-based research (i.e. "how can we represent x?" instead of "what can we represent with what we have?")

-They promote a global perspective in representation language development

Knowledge representation language bias

The field of machine learning fruitfully describes learning algorithms in terms of their inductive bias: a learner's restriction bias is a restriction on the set of hypotheses it considers and its preference bias is a preference for some hypotheses over others (Mitchell, p. 64). Analogous forms of bias can be used to informally characterize knowledge representation languages, except that instead of considering the space of hypotheses, we consider the space of knowledge to represent. The "restriction bias" of a knowledge representation language is given by the concepts it can represent; its "preference bias" is given by those it concisely states and quickly processes. The utility of this concept was recognized as early as McCarthy (1959): "If one wants a machine to discover an abstraction, it seems most likely that the machine must be able to represent this abstraction in some relatively simple way."

Since most knowledge representation languages are very expressive in principle (production systems and programming languages are Turing equivalent, so they can represent *anything* in principle), the more relevant issue is what a given language can represent easily in practice – its preference bias. As Russell and Norvig point out, standard programming languages cannot easily represent many logical propositions like "There is a pit in [2, 2] or [3, 1]" (p. 241).

Arguably, the goal of a knowledge representation language should be to match the bias of the human internal representation language (often referred to as "mentalese").

Assuming plausibly that the bias of this language is based on the principle behind many compression algorithms (such as the Huffman encoding) – that frequently encountered items should be given a preferred representation – we can estimate the bias of mentalese (or at least the declarative side of it) by looking at the frequency of knowledge types in English. Indeed, arguably the ideal bias would be based on this whether or not mentalese is. Identifying the frequency of knowledge types (e.g. nested quantifiers, disjunctions, uncertainty, etc.) in a random sample of English sentences would be a good start. For instance, given that a large proportion of verbs are action verbs and not state-of-being verbs, one is led to wonder why actions are not primitives in first-order logic and many other languages. Benchmark problem sets should also incorporate this bias – either by having problems that reflect the actual distribution or by weighting them appropriately.

Finally, the bias of a knowledge representation language, like the bias of a machine learning algorithm, need not be static. Mentalese certainly is not: cognitive development traces the evolution of the human representation language as humans acquire and compress increasingly abstract representations that consequently extend the boundary of the subtlety of concepts that can be practically represented. Explanation-based learning (Mitchell, p. 307) has been proposed as one mechanism of modifying the representational bias in machines. We discuss some other ideas in a later section.

Analysis of existing representation languages

Below, we briefly analyze the major existing representation languages. The descriptions of their respective strengths and weaknesses express their biases. There are two important observations of this analysis. The first is that the capabilities of the languages are nearly disjoint. Logics, for instance, are the most expressive declarative languages, but cannot represent the structure of knowledge as frames can, cannot represent similarity relationships as connectionist models can, and cannot represent procedural knowledge as production systems and programming languages can. The second observation is that the capabilities of the languages fall far short of covering the space of ideas that humans can represent. In particular, as observed among the insights from linguistics, these languages correspond to small subsets of English.

Logics: declarative knowledge
Summary: Expressive declarative knowledge representation lacks procedural knowledge (besides reasoning) and robustness.
Applications, examples: CYC, logic puzzles
Types: propositional logic, first-order logic, higher-order logics, temporal logics, probabilistic logics, description logics, modal logic
Strengths: expressive formal declarative knowledge that is static, certain, and consistent
Weaknesses: procedural knowledge (e.g. addition), procedural bias (context-dependent heuristics, indexing issues, search control), knowledge structure, similarity and distributed representations of common features, uncertain, approximate, vague, or inconsistent knowledge, context, time and change, beliefs, higher-order concepts, extendibility and modifiability

Semantic networks / frames: categories

Summary: Specialized representation of categorical knowledge allows for natural conceptual organization and fast inheritance inference but lacks the expressiveness of full logics.
Strengths: declarative, certain, static categorical knowledge and inheritance
Weaknesses: continuous or degree variables (age, weight, etc.), the full power of logic (negation, disjunction, existential quantification, nested functions), uncertainty, dynamic worlds, procedural knowledge, other limitations of logics

Connectionism: patterns
Summary: Computational model of associationism excels at low-level pattern recognition and similarity but inadequately represents serial procedures and abstract knowledge.
Applications, examples: ALVINN, optical character recognition, generally classification and regression, perception and control
Types: perceptrons, backprop, radial basis, competitive and Kohonen, recurrent backprop, Hopfield and Boltzmann, Helmholtz, Elman
Strengths: pattern recognition, similarity, graceful degradation, associationism, robustness to noise
Weaknesses: serial procedures, abstract knowledge (relations, quantification, individuals, recursion, rules, compositionality), rules, black box

Rule-based (production) systems: control
Summary: Powerful representation of control lacks the declarative knowledge power of FOL and the procedural organization of programming languages.
Applications, examples: cognitive modeling (solving algebra problems, video game agents), expert systems (especially medical diagnosis (DENDRAL, MYCIN))
Types: cognitive modeling (ACT-R, SOAR, EPIC, 4CAPS), expert systems (CLIPS), with or without frames
Strengths: control, procedural knowledge, procedural bias
Weaknesses: declarative knowledge, behavior with lots of declarative knowledge, context-independent knowledge, pattern recognition, abstract knowledge, modularity (subprocesses), sequential processes, proliferation of rules, indexing, uncertainty, extendibility and modifiability

Programming languages: serial procedures
Summary: Expressive languages lack support for declarative and procedural knowledge structures.
Applications, examples: operating systems, software
Types: functional (SML), imperative (C), concurrent, object-oriented (C++, Java)
Strengths: abstract procedures, sequential processes, recursion, expressiveness subsumes all others
Weaknesses: poor language bias – little support for the declarative knowledge structures needed

Probabilistic graph models: uncertain state

Summary: Probabilistic model of state structure is weak on certain declarative knowledge and all types of procedural knowledge.
Applications, examples: medical diagnosis (Pathfinder) and other expert systems, causal reasoning
Types: Bayesian networks, Dynamic Bayesian networks
Strengths: (conditional) independence relationships among variables
Weaknesses: limited expressiveness – propositional variables, unknown probabilities may be required, procedural knowledge

Finite state machines / Markov models: (uncertain) state transitions
Summary: (Probabilistic) model of state transitions lacks state and action structure and incorporation of prior knowledge.
Applications, examples: speech recognition, control
Types: FSMs, Markov chains, MDPs, HMMs, POMDPs, Reinforcement Learning
Strengths: simple representations of state transitions, utilities of actions
Weaknesses: unstructured states and actions, incorporation of prior knowledge, intractability of large problems, fixed temporal resolution

Part II: Cognitive System Design

Introduction

This part of the project aimed to develop a knowledge representation language that integrated as many types of knowledge as possible. The design of the language drew upon all major existing knowledge representation languages (frames, production systems, first-order logic, programming languages, connectionism, and probabilistic graphical models). The original intent was to develop the language and a small set of benchmark problems covering the major types of knowledge in parallel in a series of incremental design cycles. Time limitations, however, prevented us from completing multiple design cycles, so we have only the first set of problems and the first language specification. These are described in the following sections.

The Breakfast Benchmark Problems

For the purposes of designing a knowledge representation language, the benchmark problems developed in the first part of this project were too many and too broad for guidance. Since the goal of this part of the project was to develop a language that covered as many types of knowledge as possible, we developed a shorter set of problems to focus on. We developed a few problems for each type of knowledge, and made them all in the same commonsense domain of breakfast to force any knowledge representation language handling all of the problems together to integrate the different types. The entire set of problems is given in the Appendix. Here are a few representative problems:

-Some people eat five small meals a day. (relations)

-Breakfast is usually smaller than dinner. (uncertainty)
-If people are hungry and eat enough, they are no longer hungry. (change)
-In China, rice is common for breakfast. (context)
-Compare breakfast and lunch. (pattern recognition)
-How many meals does a person eat in a year? (problem solving)

Cognitive System Specification

To develop a new knowledge representation language (which we will refer to as a "cognitive system"), we drew upon two main sources for guidance: existing knowledge representation languages and breakfast problems for the major types of knowledge. The new language aspires to the declarative expressiveness of logics, the conceptual clarity of frames, the fine-grain control of production systems, and the abstract procedural organization of programming languages, in addition to some coverage of all major types of knowledge as represented by the breakfast problems.

The system is intended to be interactive, with a dynamic knowledge base. Components of the system are divided into constructs for declarative knowledge and for procedural knowledge.

Declarative knowledge is comprised of objects, functions, propositions, sets, and abstract sets. Declarative knowledge is about objects, such as "Tim", "three", "age(John)", and "clocks". As in logic, functions map objects to other objects, providing a way to reference and discuss objects without naming them. For instance, we can discuss "head(John)". Propositions are the units of declarative knowledge. Ground propositions make assertions about the relations among particular objects. We might assert "(smaller breakfast dinner)" or "(equals age(John) 22)". Sets, such as "people" or "squares", are given special treatment, so that all objects are organized into a set hierarchy organized by special relations such as "equals" and "subset". Abstract sets are sets defined by a property which is given by a proposition with a single variable. This allows for the discussion of sets such as "breakfasts in China" or "people younger than 21 at the party". Sets serve to organize knowledge, to allow for quantified assertions, and to enable automatic inference by inheritance.

Procedural knowledge is organized around tasks and agents. The user requests the completion of tasks (e.g. answering a question or sorting a list), which are done by appropriately selected agents that may request subtasks themselves. Agents have three components: a task pattern, a set of condition propositions, and a sequence of task requests. An agent is applicable to a given task if its task pattern matches and its condition propositions are satisfied. If chosen for execution (based on utility), it executes its sequence of task requests. Matching condition propositions makes use of inheritance within the set hierarchy: a condition proposition is satisfied if it can be inferred by inheritance. Finally, ask and tell are special tasks that respectively query and modify the global knowledge base. Agents for other tasks access and use knowledge through these task requests and through condition propositions.

This specification is considered a working draft; as we begin to implement it, there will undoubtedly be some number of modifications and extensions.

Declarative knowledge

Objects

Syntax cases

1) Name: character string (e.g. John, smaller, box23, etc.)
2) Abstract set (see toSet)
3) Function of n objects of the form f(x_1,…,x_n)

Notes

-Functions, relations, and tasks are also objects

-Hence one can make assertions about them, they may be organized into categories, and can be pattern matched upon

Exs:

John, father(John), father(father(John)), children(Tim, Ann), father, lessThan, sort

Object patterns

Syntax cases

1) Object
2) Variable (character string preceded by '=')
3) Function with arguments that are object patterns
4) Function variable with arguments that are object patterns

Exs:

=x, father(=x), children(Tim, =x), father(father(=x)), =f(John)

Propositions

Syntax

1) (relation object-1 … object-n)
2) not(1)

Notes

-A relation is just another object; it is only distinguished by its role in the current proposition

Exs:

(smaller breakfast lunch)
(equals father(John) Tim)

Sets

Special relations

-Unlike most relations, these are NOT explicitly stored in the system – instead, they have direct effects upon the internal representation of the knowledge base

equals

-The system unifies the two given names, pooling the propositions in which they occur, and listing both names as synonyms

-Exs:

(equals father(John) Tim)
(equals toSet(=x, red(=x)) toSet(=x, (equals

color(=x) red)))

equals*

-For a new object, this is the same as equals
-If this object is currently a synonym for another object, then the object is freed from that association and reassigned to the new given object
-Note the relationship to pointers
-This is useful for temporary objects

delete

-An object name may be deleted if it is a synonym for another object
-This is useful for temporary objects

member, subset

-The system adds the object to the set hierarchy

disjoint, partition

-These are important for organizing subsets
-Unless the system knows otherwise, it assumes that function range elements are disjoint, so that functions naturally partition sets

Special functions

intersection, union, complement

-These map sets to a new set

size

-This gives the number of members of a set

prop

-This gives the proportion or relative size
-Ex:

(equals prop(men, people) 1/2)

-This has the obvious relation to the sizes in the case of finite sets
-This is useful for probabilistic knowledge

Abstract sets (toSet)

Syntax

toSet(variable, [proposition that may use the variable *once* in place of any object])

Notes:

-Restrictions

-The variable may only appear once in the defining proposition – this simplifies the set hierarchy, improves tractability, and avoids paradoxes (such as Russell's

paradox: toSet(=x, (member =x =x)), (ask (member =x =x) =ans)?)

-Nesting of abstract sets is not permitted (the proposition pattern used to define an abstract set may not contain another abstract set)

Exs:

"People who make less than $20000 in a year"

intersection(toSet(=z, (lessThan income(=z) 20000)), people)

"Half of the class received passing grades"

(equals prop(intersection(class, toSet(=z, (member grade(=z) passingGrades))), class) ½)

"Bipeds are animals with two legs."

(equals bipeds intersection(animals, toSet(=z, (equals size(legs(=z)) 2))))

"All people have large intestines."

(subset people toSet(=z, (hasPart =z largeIntestine)))

-Further notes

-Details of maintaining the abstract set hierarchy must be determined

-Often the abstract set hierarchy will mirror the set hierarchy

-Argument for feasibility

-Size: linear function of number of "tells"

-Time: should only need to consider other sets involving the same proposition and objects – locality

Numbers

-As with many systems, the system will have special support for numbers

-In particular, all numbers are defined as objects

-Basic arithmetic

-add, subtract, mult, divide, etc.

-Comparison operations

-lessThan, greaterThan, etc.

Procedural knowledge

Condition proposition patterns

Syntax

1) (relation-pattern object-pattern-1 … object-pattern-n)
2) not(relation-pattern object-pattern-1 … object-pattern-n)
3) NOT (1) or (2)

Ex:

(older father(=x) mother(=x))

(smaller =x elephant)

Notes

-Type (2) is matching on the logical negation of the proposition (if the system knows that the negation is true)
-Type (3) performs a negative match (if the system doesn't know that the proposition is true)
-Proposition patterns are used for conditions for agents
-A pattern is matched if the system explicitly "knows" the proposition, using only inheritance for inference (see the next section)
-Negative patterns (beginning with NOT) match if the system does not "know" that the proposition is true by inheritance
    -Note that the proposition may still be true!
    -This is similar to negation as failure
-Patterns serving as conditions for agents may introduce new variables, which serves to retrieve related objects
-Negative patterns for conditions for agents may not contain new variables in them

Inheritance-based pattern matching

-Pattern matching uses a transparent inheritance process and the set hierarchy, so that a proposition pattern is matched by a set of objects if the proposition can be inferred directly through inheritance
-Ex
    -Suppose we have
        (member John people)
        (subset people animals)
        (subset animals toSet(=z, hasPart =z head)))
    -Then suppose we have the proposition pattern
        (hasPart =x head)
    -John will match =x for this proposition pattern
-Hence the set hierarchy provides knowledge that is immediately accessible through inheritance
-For more complex questions, agents become involved
-For purposes of efficiency, inheritance-based pattern matching is optional – any proposition condition followed by a "#" will be matched without inheritance

Task request

Syntax

(taskObject arg-1 … arg-n)
where taskObject is a task object and arg-i may be either an object or a proposition

Notes:

-This definition is very similar to the definition of a relation, except that propositions may be included as arguments
-Tasks are objects that may be organized into hierarchies, etc.

Exs

(compute prod(3,4))
(compare breakfast lunch)
(sort list34)
(findPath Rome Paris)
(tell (member John people))
(ask (member John people) answer)

Task pattern

Syntax

(taskPattern arg-pattern-1 … arg-pattern-n)
where each arg-pattern is either an object pattern or a positive proposition pattern (without a NOT)

Exs

(compute =x)
(compute prod(=x, =y))
(ask (member =x people) =y)
(sort =x)

Agent

Syntax

agent taskPattern
conditions
    propositionPattern-1
    …
    propositionPattern-n
actions
    taskRequest-1
    …
    taskRequest-m

Notes

-The proposition patterns may introduce new variables, which can be used to retrieve related objects
-"Return" values are included in the taskPattern, as in pass by reference
-Agents that pattern match on the task must have an associated task type such that they are only applicable to tasks of that type

Exs

/* forward search using transitivity */
agent (ask (=r =x =y) answer)
conditions
    (transitive =r)
    (=r =x =z)
actions
    (ask (=r =z =y) yes) /* only succeeds if "yes" */

(tell (equals answer yes))

Agent Instance

-Each possible way to match the set of proposition patterns for the given task yields an agent instance

Control cycle

-Task types: all tasks are one of two types: universal or existential

-Control is different in these two cases

-For universal tasks, all matching agent instances are executed

-For existential tasks, we just want to find one agent that can complete the task

A) Universal tasks

1) Find all matching agent instances
2) If all matching agent instances have been executed, the task is completed
3) Otherwise, choose the agent instance with highest utility that has not been executed and execute it
4) Go back to (1)

B) Existential tasks

1) Find all matching agent instances
2) If there are no matching agent instances that have not already been tried, then we fail to complete the task
3) Otherwise, choose the agent instance with the highest utility and execute it
4) If the agent instance completes, the task is completed
5) Otherwise, go back to (1)

-Notes

-In both cases, the set of agent instances may change as a result of the execution of an agent, since the knowledge base can be modified

Task request success

1) An agent instance succeeds if its sequence of task requests succeed
   -After the first task request that fails, the agent instance is aborted
2) An existential task request succeeds if some matching agent instance succeeds.
3) A universal task request succeeds if all matching agent instances succeed. In particular, if there are no matching agent instances, it succeeds.

Temporary objects

-Agents may use objects only temporarily to keep track of the state of computation

-These are just ordinary objects, except in how they are used:
    -They can only be assigned to other objects
    -They may also be re-assigned (with "equals*")
    -They may be deleted (with "delete")
-Effectively, they are just point to permanent objects to record the state of computation

Special tasks

tell

Executed as follows:
1) The system adds the proposition, restructuring the set hierarchy as necessary
2) The task request is made as a universal task, allowing for "if-added" types of effects

ask

Executed as follows:
1) The system attempts to answer the question directly via pattern matching and the set hierarchy
2) If it fails, then the task request is made as an existential task, allowing for agents to answer the query

associate

Displays all propositions associated with an object and all of those that can be derived directly through inheritance. This is all "immediately available" knowledge about the object – the knowledge that is available for pattern matching conditions.

Interface

-The user makes task requests the same way any agent does

Retrieval and Indexing

-Each object points to each instance in which it occurs in a relation

Comments

-Borrowed from C, text between "/*" and "*/" is ignored.

Analysis

Coverage of key features of other languages

1) Logics: declarative expressiveness

Declarative expressiveness is achieved through the use of propositions, functions, and abstract sets. Explicit representations of propositions and functions both significantly increase the range of possible assertions over standard frame and rule based systems.

Abstract sets further empower the system to represent relations in a structured way. They correspond to noun phrases in English (e.g. "people who make less than $20,000 a year") and draw upon the ideas of description logics. Moreover, they come in useful for organizing uncertain knowledge (discussed later).

That said, the language cannot declaratively represent all first-order logic statements. On the other hand, the language can represent certain types of higher-order statements that cannot be represented within first-order logic and can represent other types of facts more easily. Its representation of the structure of declarative knowledge and of procedural knowledge are far more extensive than first-order logic. The coverage of logical constructs and possible extensions are discussed below.

Examples

"Every male dog is named Spike."
(subset intersection(males, dogs) toSet(=z, (equals name(=z) Spike)))

"More than 1000 people have a million dollars."
(equals millionaires intersection(people, toSet(=z, (greaterThanOrEqual
wealth(=z) dollars(1000000))
(greaterThan size(millionaires) 1000)

Implementation of a simple backward chaining verification system

-Facts
-These are of the logical form p(x_1,x_2, …, x_n) with all variables bound
(p x_1 x_2 … x_n)

-Rules
p1(x_1, … , x_n) and … and pm(x_1, … x_n) => q(x_1, …, x_n)

agent (ask (q =x_1 … =x_n) answer)
conditions
actions
(ask (p1 =x_1 … =x_n) ans1)
(ask (p2 =x_1 … =x_n) ans2)
…
(ask (pm =x_1 … =x_n) ansm)
(tell (equals answer and(ans1, … ansm)))

-Performs depth-first backward chaining verification
-Could easily be modified to shortcircuit results
-More difficult extension: arbitrary satisfiability
-But we claim that general purpose satisfiability is a bad idea: usually more controlled, problem specific search methods are desired to find paths, sort lists, etc.

-Note that conditions are not exploited and neither is the set hierarchy

Coverage of Logical Constructs (constructs based on Russell, p. 247)

Objects

Represented by objects

Functions

Represented by functions

Predicates

Represented by relations

Connectives

Negation

Represented in a limited form by not, possibly combined with abstract sets:

```
/* John is not a dog */
not dog(John)
not(member John dogs)

/* People don't have tentacles */
not hasPart(people, tentacles)
not(subset people toSet(=z, (hasPart =z tentacles)))
```

The NOT modifier to conditions can also serve as a weak form of negation (if the condition is not known to be true).

Conjunction

Represented by intersections of sets and by agent conditions, which are treated conjunctively.

Disjunction

The disjunction of arbitrary propositions cannot be represented naturally within the system. However, the disjunction of particular objects of propositions can be represented through a set of possibilities:

```
/* "Laura loves either Mark or Sam." */
(loves Laura (Mark or Sam))
(loves Laura mysteryMan)
(member mysteryMan possibilities)
(member Mark possibilities)
(member Sam possibilities)
(equals size(possibilities) 2)
```

Syntactic sugar could simplify such representations.

Implication

Implications with a single universally quantified variable and both hypotheses and conclusions containing conjunctions of positive or negative predicates with only a single occurrence of that variable can be represented using the abstract set hierarchy:

/* John buys every gadget he wants */
forAll x wants(John, x) and gadget(x) => buys(John, x)
(subset intersection(gadgets, toSet(=z, (wants John =z))) toSet(=z, (buys John =z)))

Implications with a similar form, but possibly multiple quantified variables can be represented procedurally by question-answering agents with appropriate conditions and actions.

Equivalence

Equivalence can be used in the same situations implication can.

Equality

Represented by the "equals" relation.

Quantifiers

Universal

Universal quantification is over the members of a set, arguably the most frequent way it arises in practice. Universal quantification of multiple variables must be handled procedurally by question-answering agents.

"John likes all deserts."
forAll x, desert(x) => likes(John, x)
(subset deserts toSet(=z, (likes John =z)))

Existential

There is no explicit support for existential quantification: such objects must be instantiated, referenced by functions, or be implicit members of a set.

"John owns a watch."
thereExists w, watch(w) and owns(John, w)
(member watch23 intersection(watches, toSet(=z, (owns John =z))))
(member watch(John) intersection(watches, toSet(=z, (owns John =z))))

"Some people like Sam."
thereExists x, person(x) and likes(x, Sam)
(greaterThan size(intersection(people, toSet(=z, (likes =z Sam)))) 0)

Nested quantifiers

Nested quantifiers cannot be represented naturally.

2) Frames: conceptual organization

This system achieves conceptual organization primarily by maintaining a set hierarchy for objects and by maintaining references from all objects to each instance in which they occur in each relation. As a result, it is easy to retrieve all "immediate" facts about an object, which are those facts that are directly asserted or can be inferred by inheritance (i.e. those that can be used for pattern matching). This is the "associate" task, which can be useful for designing and analyzing a system.

Compared to frames, this system has a more flexible representation of declarative knowledge since frames correspond to sets of propositions. The procedural structure is also more flexible, since it subsumes procedural attachment. An implementation of a simple frame system (as described in Brachman & Levesque, Ch. 8) is shown below.

Implementation of a simple frame system

-Generic frames, individual frames
    -Generic frames are sets, individual frames are objects
-Slots-value pairs for individual frames
    -Represented as propositions of the form (equals f(x) y)
        Ex:
        "John's age is 22."
        (equals age(John) 22)
-Slots-value pairs for generic frames
    -Represented using abstract sets to quantify over members
        Ex:
        "Bananas are yellow."
        (subset bananas toSet(=z, (equals color(=z) yellow)))
-ISA relation
    -Represented by "subset"
-INSTANCE-OF relation
    -Represented by "member"
-Inheritance
    -Automatically done through the set hierarchy
-IF-ADDED effects
    -Implemented as "tell" agents
-IF-NEEDED effects
    -Implemented as "ask" agents
-Defaults and overriding
    -See the section on knowledge maintenance

3) Production systems: fine-grain control

Fine-grain control is achieved through agents, which may be specialized based not only by task patterns but also based on knowledge in the form of proposition conditions. The control structure allows for multiple competing strategies at the level of abstraction of any agent. Hence there may be competing agents for even sub-sub-sub-tasks if appropriate.

Compared to production systems, this system provides much greater support for declarative knowledge representation in the form of propositions and for abstract procedural control through the sequential task request structure of agents. Agents represent a balance between the rules of production systems and the functions of programming languages.

Examples

-Context-specific control of problem solving

```
/* suppose these propositions have been told to the system */
(smaller breakfast lunch)
(smaller lunch dinner)
(transitive smaller)

/* forward search using transitivity */
agent (ask (=r =x =y) answer)
        conditions
                (transitive =r)
                (=r =x =z)
        actions
                (ask (=r =z =y) yes) /* only succeeds if "yes" */
                (tell (equals answer yes))

/* backward search version of using transitivity */
agent (ask (=r =x =y) answer)
        conditions
                (transitive =r)
                (=r =z =y)
        actions
                (ask (=r =x =z) yes) /* only succeeds if "yes" */
                (tell (equals answer yes))
```

-Implementation of a simple production system (as described in Brachman and Levesque, Ch. 7)

-Working memory elements

-Represent as objects, with a set of propositions for their slot-value pairs

-Rules

-Agents with a dummy task

-Rule conditions -> agent conditions

-Rule actions are a sequence of tell commands followed by the dummy task

-ADD working memory element

-MODIFY working memory element

-We need to have appropriate inconsistency handling for MODIFY

-What about REMOVE? Less simple …

-ACT-R doesn't have a remove either

-Note no use of task structure!

4) Programming languages: abstract procedural organization

Agents are the unit of procedural organization. Sequential procedures can naturally be represented with agents and procedural knowledge is organized into tasks and subtasks. Another nice feature of the system, not found in standard programming languages, is the clean separation of task specifications from the agents that perform them. This provides for modularity, so that the agents that perform tasks can be replaced and new agents for that handle special cases better can be added. As shown in the examples below, this language is powerful enough to implement such procedures as a sorting algorithm and covers most standard programming constructs.

The most important difference between this system and standard programming languages is in its data structures. Instead of having many data structures, this system uses propositions as the universal data structure: all information about all objects is represented in terms of propositions. This provides for a more fluid architecture. In particular, agents need not make as extensive assumptions about the data structures used to represent the objects they process, since they can be assumed to take the form of propositions. Another difference between this system and programming languages is the support this system provides for pattern matching and problem solving.

-Example

```
/* Compute the calories of a meal. */

/* Suppose we have told the system these facts */
(equals calories(burger) 300)
(equals calories(fries) 200)
(equals calories(coke) 100)
(member burger parts(meal23))
(member fries parts(meal23))
(member coke parts(meal23))

/* If you already know the calories, simply retrieve them */
agent (compute calories(=x) =ans)
        conditions
                (equals calories(=x) =y)
        actions
                (tell (equals =ans =y))

/* Decompose the calorie computation into the sum of the calories of its
        parts */
agent (compute calories(=x) =ans)
        conditions
                (greaterThanOrEqual size(parts(=x)) 1)
        actions
                (tell (equals* =ans 0))
```

```
                    (sumParts calories =x =ans)

        /* sumParts is a universal task */
        /* sum the value of =f over all parts of =x */
        agent (sumParts =f =x =ans)
                conditions
                        (member =y parts(=x))
                actions
                        (compute =f(=y) tmp)
                        (compute sum(=ans, tmp) tmp2)
                        (tell (equals* =ans tmp2)))

/*****************************************/
/* selection sorts a doubly-linked list */

/*  for head, prev(head) = null */
/*  for tail, next(tail) = null */

/*  num(listItem) is the number in this entry */
/*  next(listItem) is the next item */
/*  prev(listItem) is the prev item */

/* selection sorts a doubly-linked list =x */
agent (sort =x)
        conditions
                (member =x listItems)
        actions
                (tell (equals* =tmp infty))
                (findMin =x =tmp)
                (removeItem =tmp)
                (clearItem =tmp)
                (addItemToFront =x =tmp)
                (sortNext =x)

/*  universal task, proceeds if the next item is not null */
agent (sortNext =x)
        conditions
                NOT (equals next(=x) null)
        Actions
                (sort =x)

agent (findMin =x =min)
        conditions
                (member =x listItems)
        actions
                (updateMin num(=x) =min)
```

```
                (findMinNext =x =min)

/* universal task, updates the min if necessary */
agent (updateMin =num =min)
        conditions
                (lessThan =num =min)
        actions
                (tell (equals* =min =num))

/* universal task, proceeds if not done */
agent (findMinNext =x =min)
        conditions
                (member =x listItems)
                NOT (equals next(=x) null)
        actions
                (findMin =x =min)

/* universal task, removes previous pointer */
agent (removeItem =x)
        conditions
                (member =x listItems)
                NOT (equals prev(=x) null)
        actions
                (tell (equals* next(prev(=x)) next(=x)))

/* universal task, removes next pointer */
agent (removeItem =x)
        conditions
                (member =x listItems)
                NOT (equals next(=x) null)
        actions
                (tell (equals* prev(next(=x)) prev(=x)))

/* nullifies prev and next pointers */
agent (clearItem =x)
        actions
                (tell (equals* prev(=x) null))
                (tell (equals* next(=x) null))

agent (addItemToFront =y =x)
        conditions
                (member =y listItems)
                (member =x listItems)
        actions
                (tell (equals* next(=x) =y))
                (tell (equals* =y =x))
```

Coverage of standard programming constructs (based on Kernighan & Ritchie).

-If, else, switch
    -Represented by pattern matching
-Types, structs, classes
    -Represented by sets
-Functions, blocks
    -Represented by agents
-Exception handling
    -Postponed, see the extensions section
-Local variables
    -Can be implemented in a limited fashion by objects
    -Discussion of temporary instantiation????????
-Global variables, constants
    -Represented as objects
-Pointers
    -Functions, objects
-Recursion
    -Represented by agent instances that make requests that other instances of the same agent handles
-Return values
    -Answers are passed by reference – return object included in the task description
-While loops
    -An agent for a universal task corresponding the while loop body with conditions corresponding to the while loop condition

5) Connectionism: patterns and similarity

The principal benefits of connectionist systems for representation of high-level concepts are their support for patterns and similarity relations and their consequent robustness to errors (Anderson, p. 33).

The precise relationship between conceptual similarity (instance theories) and categories (abstraction theories) is a topic of some debate (Anderson, p. 164). We hypothesize that similarity is used only for small categories – so that after someone sees their first bear, subsequent bears are compared to it until some abstract category of bears emerges. Although the literature appears to focus on concrete objects like "dogs" and "birds" for this debate (Anderson, p. 164), we argue that it is enlightening to also consider more abstract concepts, such as math problems. As in the case of "bears", we would hypothesize that the first few instances of a new type of math problem are solved by analogy with examples until some abstract categorical representation of the problem class emerges.

Our cognitive system supports patterns and similarity relations through the use of the set hierarchy and inheritance-based pattern recognition for agents. Because of the structure of the set hierarchy, it is easy to identify the properties that two objects have in

common. Also it is easy to form new categories based on arbitrary sets of properties. Automatically forming such categories when useful is an area for future research.

Anderson (pp. 33-34) identifies several intelligent capabilities of connectionist systems: solution of what we call the "inverse dictionary problem," fuzzy retrievals, and categorical generalization. We briefly sketch how simple extensions to our system enable it to have each of these capabilities:

Inverse Dictionary Problem

The "inverse dictionary problem" is the problem of efficiently retrieving an object based on a description of it. Anderson's example is answering "Who do you know who is a Shark and in his twenties?" (p. 33). Within our system, such descriptions naturally correspond to intersections of abstract sets. The following proposition pattern will match any such individuals:

```
(member =x intersection(toSet(=z, (member =z Sharks)),
                        toSet(=z, (member age(=z) interval(20, 29)))))
```

Fuzzy retrieval

The fuzzy retrieval problem is like the basic inverse dictionary problem, but involves finding the closest match in the case of no exact match. This could be implemented easily within our system by traversing up the set hierarchy to increasingly general sets (increasing the "fuzziness" of the match) until a match is found.

Categorical generalization

Anderson's example of this capability of connectionist models is their ability to identify that "Jets tend to be single, in their 20s, and have a junior high education" from examples (p. 34). Within our system, this capability would be implemented through the use proportion (or probability) estimation, as discussed in the Extensions section. In short, the observed proportions of the subsets of interest are counted. Hence the system would know that 5/9 of Jets are single, 5/9 are in their 20s, and 2/3 have a junior high education (using the data in Anderson p. 32).

As another simple example of the capability of this system to handle, we present a simple analogy problem solved by the system. The problem is Jupiter : Sun :: Moon : ?.

```
/* These assertions are in the knowledge base */
(orbits jupiter sun)
(orbits moon earth)

/* The task request is: (solveAnalogy jupiter sun moon ans) */
/* This is an extremely simple analogy solving agent using pattern matching */
agent (solveAnalogy =x1 =y1 =x2 =unknown)
        conditions
                (=r =x1 =y1)
                (=r =x2 =y2)
        actions
```

```
(tell (equals* =unknown =y2))
```

Other features and discussion

Uncertainty

Probabilities are represented by proportions of sets. The special set constructs for partitions, complements, disjointedness, etc. can naturally extend to probabilistic assertions about distributions. The use of abstract sets further extends probabilistic statements to properties.

Examples

```
/* Half of all people are males. */
(equals prop(males, people) ½)

/* In China, rice is common for breakfast. */
(equals commonConst 0.75)  /* something is common if its proportion is at least 0.75 */

/* Of all breakfasts in China, at least 0.75 contain rice */
 (greaterThanOrEquals
prop(intersection(breakfasts,
        toSet(=z, (equals location(=z) China)),
        toSet(=z, (contains =z rice))),
intersection(breakfasts,
        toSet(=z, (equals location(=z) China)))
commonConst)
```

Higher-order concepts

Higher-order concepts come from the reification of both functions and relations. Hence one can assert, for instance, that a function is monotone or that a relation is commutative. Here is a simple example that exploits the monotonicity of a function:

```
/* reduce determining f(x) < f(y) to determining x < y in the case that f is monotone */
        agent (ask (lessThan =f(=x) =f(=y)) =ans)
                conditions
                        (monotone =f)
                actions
                        (ask (lessThan =x =y) yes)  /* succeeds only if the answer is yes */
                        (tell (equals* =ans yes))
```

This is a limited form of higher-order knowledge. See the section on extensions for discussion of further possibilities.

Dynamic problem representations

This system naturally allows for dynamic problem representations. Given a new task, an agent can modify or extend the given task description, or translate it into another representation that can be used by other agents.

Similarly, if the system is attempting to solve a problem and tries an approach that ultimately fails, it may learn more about the problem from the failed attempt and may store partial solutions that can later by used by another agent. Specifically, the agents that fail to solve the problem may modify and extend the knowledge base. When the system backtracks to find other possible problem solving strategies, it re-matches agents and may find that new strategies are applicable because of the changes and additions – what it learned about the problem from its failed effort.

Arguably, this is an important feature of human problem solving that is significantly more difficult to represent in other systems. Standard programming languages like Java and C have static data structures, so that functions may translate from one data structure to another, but cannot modify data structures at runtime for the purposes of a particular instance. The use of propositions as universal data structures allows for fluid representations when appropriate. Similarly, standard problem solvers that represent problems as static search spaces cannot represent "higher-level" operators that modify the state space structure itself.

References: declarative, procedural, and communicative perspectives

Functions in logic serve to make assertions about unnamed objects (e.g. leftHandOf(father(John))) and to simplify the representation of certain types of relations among objects. In programming languages, functions are evaluated or executed and pointers refer to other objects. Hence "mult(23, 54)" evaluates to "1242" and "john->father->leftHand" references John's father's left hand. Logical functions are thus similar to pointers, except that they may use multiple arguments for the purposes of dereferencing (e.g. children(x, y)). Finally, natural languages like English have the most flexible means of referencing objects. Logical functions such as "children(Tim, Ann)" can be stated simply in English using "of" and conjunctions ("the children of Tim and Ann"). The structure of noun phrases in English, however, is much more general, and allows for many references that do not have a natural function decomposition (e.g. "the hot girl we saw at the beach yesterday") and that more generally may depend upon the current context. Indeed, for the purposes of communication, references need only be precise enough for the listener to uniquely identify the intended object.

This system integrates ideas from all of these perspectives. Functions may be used as logical functions to make assertions about unnamed objects:

```
/* "The highest grade on exam 2 was less than the highest grade on exam 1." */
(lessThan max(grades(exam2)) max(grades(exam1)))
```

Moreover, the identity of functions can be resolved: they may be evaluated as in programming languages. The following agents can be used to compute the maximum of the known members of a set (and, in particular, the highest grade on exam 2). Note that this automatically memoizes the answer (which will happen any time the task is to resolve the identify of a functionally defined object in the knowledge base):

```
/*  This general agent retrieves values in the case that they are already known */
/*  It is given high utility so it is preferred to re-computation */
agent (compute =x =ans)
        conditions
                (equals =x =y)  /* answer is already known */
                (member =y names)  /* =y is an object name, not a function */
        actions
                (tell (equals =ans =y))

/* Organize computation of the max */
agent (compute max(=x) =ans)
        actions
                (tell (equals* =ans negInfty))
                (compareAll =x =ans)
                (tell (equals max(=x) =ans))

/* Compare each member of =x to =max and update =max accordingly */
/* compareAll is a universal task */
agent (compareAll =x =max)
        conditions
                (member =y =x)
                (greaterThan =y =max)
        actions
                (tell (equals* =max =y))
```

Finally, the system distinguishes between communication and internal representation as humans (and many other systems) do through a layer of interpretation. Hence, the system internally represents "father(John)" and "Tim" as synonyms for the same object, so that if it is told any assertion involving either, the resulting internal representation is identical. A special function "toMember" could easily extend the system to simplify the representation of noun phrases in the case that a given set has a unique member:

```
/*  "The clown with the blue nose is funny." */
/*  (assuming the system knows only one such clown) */
(funny toMember(intersection(clowns, toSet(=z, (equals color(nose(=z)) blue)))))
```

Universal propositions with context-specific application

Representing propositions in a strictly declarative form results in universality: facts are not tied to any particular problem or setting and can hence be used for new problems and situations that may arise. The lack of knowledge about exactly how and when to use facts, however, generally leads to inefficiency (Brachman & Levesque, p. 99)

Procedural representations of knowledge can incorporate heuristics and context information to make the best use of what is known (Brachman & Levesque, p. 99). The lack of context-independent representations of facts, however, can lead to redundant representations of factual knowledge.

This system gets some of the benefits of both approaches. Facts are explicitly represented as propositions which are not directly connected with any particular context, resulting in universal applicability. Facts are used by context-specific agents, however, resulting in efficient context-dependent heuristics.

We propose to replace universality with generality. Instead of having facts entirely detached from context (i.e. universal), this system aims to attach them to appropriately general contexts. As noted before, agents in this system may have widely varying generality, depending both upon their task patterns and upon their conditions. If a fact is applicable in a general context, then a general agent can use it.

Procedure execution vs. question answering vs. problem solving

Different systems have taken slightly different perspectives on exactly what a system should do. The primary dimension of difference is the amount of search involved. Programming languages generally involve the least search: programs execute specific procedures (or evaluate functions). Sorting makes for a nice procedure. Finding a solution to the 15-puzzle is somewhat less natural.

Logical systems answer questions (Russell & Norvig, p. 195). This perspective is natural for answering "Is John mortal?", but less natural for the purposes of sorting ("is there an ordered permutation of this list?"; this example is taken from a lecture by Prof. Avrim Blum). That is, satisfiability questions that can be efficiently answered by known algorithms are better framed as tasks for procedures.

Many other AI systems such as SOAR view the world in terms of problems and search spaces. This perspective is natural for the purposes of finding a solution to the 15-puzzle, but less natural for the purposes of multiplication or sorting since the search spaces are collapsed.

These different perspectives have influenced the tasks to which these systems have been applied: programs sort, logical agents reason, and problem solvers search. A system that is to be able to do all of these things must have natural representations for each type of task.

Cognitive psychology offers an interesting perspective on these tasks by placing them on a spectrum from well-practice skills (e.g. multiplying or walking) to novel problems (e.g. playing a new game) (Anderson, p. 279). As humans learn how to solve a problem (e.g. integrating), they gradually become experts, and the associated search space shrinks.

Our system provides support for execution of procedures, inference of conclusions, and search for solutions. We take the perspective that tasks are fundamental. The task decomposition into a sequence of subtasks allows for procedures (such as sorting) to be represented naturally. By contrast, in a production system, a simple sequence of steps requires many separate rules, the explicit representation of control information (such as which step we're on), and unnecessary rule matching computations (since there is essentially a single linear control path).

Support for question answering (i.e. verifying propositions) is in several forms. First, agents may accepts propositions as arguments in addition to other objects. Second, there is the automatic answering of questions based on inheritance. This is useful for answering questions for which no real search is necessary (such as "does John have a large intestine"; this example is adapted from Willingham, p. 302). Third, in the case that

a question cannot be answered automatically by inheritance, the task of answering it is treated as any other problem and specific problem-solving agents handle it.

Problem solving is naturally supported in several ways. Decomposition of tasks into sequences of subtasks allows for problems to be decomposed into sub-problems. Moreover, the control scheme automatically backtracks in the case of agent failure. Among other methods, both forward and backward search algorithms can be naturally represented. Finally, problem solving benefits from the possibility of many applicable strategies.

Evaluation on Breakfast Benchmark Problems

Unfortunately, we did not have time to properly evaluate all of the representation languages on the breakfast benchmark problems. We did make some estimates of the performance of our system. In particular, we project that an implementation of present specification (perhaps with slight modifications) can cover at least 16 of the 23 problems (with difficulties on problems 2, 6, 7, 10, and possibly 15, 16, and 21). The problems that present the biggest difficulties involve subtle declarative relationships (like problem 2: "some people eat five small meals a day") or higher-order knowledge like beliefs (problem 10). By contrast, we would estimate that a standard production system would cover only about 10 of the 23 problems. It would have difficulties with problems 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 15, and 20, primarily because these require general declarative knowledge representations.

But these are only estimates. A proper evaluation of our system and comparison with others will require demonstrating the systems on the problems. We hope to be able to perform such an evaluation after implementing our system in code.

Further Research

Implementation and Debugging

Unfortunately, we did not have time to implement the cognitive system design in code. In the process of implementing this system, we expect to encounter some number of "design bugs" and relevant specification details we failed to foresee. When we implement and test the system on numerous benchmark problems, such as the breakfast problem set we developed, we will probably find that further modifications to the design and additional syntactic structures simplify representation further. Hence this specification is considered a "working draft." We hope to complete an implementation of this system this summer.

This first implementation will complete the first iteration of the design cycle. Afterwards, we hope to continue with further iterations of jointly developing comprehensive benchmark problems and this cognitive system. Some immediate extensions are discussed below.

Investigation of coverage of specific types of knowledge

The cognitive system has ended up being more of an architectural design than originally intended. That is, we ended up focusing on coverage of the major important abstract features of existing languages (e.g. fine-grain control and conceptual

organization) instead of coverage of the major types of knowledge (e.g. time and uncertainty). Investigation of coverage of the major types of knowledge is an important next step. Further modifications to the specification may be warranted, if it cannot represent some types of knowledge easily enough.

Knowledge maintenance: assimilation and accommodation

The cognitive system is intended to be interactive and dynamic. In the language of cognitive development, it aims to both assimilate and accommodate new information (Flavell, p. 5). There are many ways in which new or deduced knowledge may call for restructuring or modifying existing knowledge. Some have been investigated in the areas of non-monotonic reasoning and truth maintenance (Russell & Norvig, pp. 358-362). Within this system, we propose the most natural way to handle knowledge maintenance is in a fashion similar to exception-handling in programming languages: with situation-specific agents that act when new knowledge has certain relationships with existing knowledge.

Perhaps the simplest relationship is direct contradiction, e.g. if the system is told that a person is both male and female. More subtle varieties also exist. Other important types include specializations and refinements of approximations. For instance, if the system is told "David has a pet" and later that "David has a dog," it should identify the relationship between these assertions and restructure the knowledge appropriately. In particular, in this case, it would make sense to simply recall that "David has a dog" and remember that "dogs are pets." This particular example could be identified easily by exploiting the set hierarchy. More generally, the approach our system will take depends critically upon the structure given to its knowledge by the set hierarchy. We claim that methods of indexing cannot be divorced from the semantics of the concepts and facts being indexed if any large scale system is to be efficient.

Default knowledge is another area of knowledge maintenance, that involves both assimilation and accommodation. Within the current system, defaults could be implemented using probabilistic knowledge and contradiction handling. For instance, suppose the system is told that "All birds fly." Then given a new type of bird, it would automatically categorize it as a subset of the set of birds that fly. So initially, penguins would be assumed to fly. When told that penguins don't in fact fly, the system would handle the contradiction with its super-class by assuming that the more specific knowledge is correct. As a result, it would modify "All birds fly" to be "Most birds fly" and re-categorize penguins as a subset of the set of birds that don't fly. Later birds would be assumed at first to be fliers since most are. Again, they might be re-categorized.

Contexts

The current system specification organizes and indexes knowledge based on its relational structure and in particular, by a set hierarchy. For the purposes of efficient retrieval, however, contexts are probably a better way to organize memory. A context is a group of concepts that frequently co-occur, and can be viewed as a higher-level approximation to the low-level theories in cognitive psychology about activation of concepts in the brain (Anderson, pp. 181-186). For instance, the context of the beach might include the concepts of sand, sea, sun, seaweed, shells, and volleyball. The associative structure of concepts appears not to be identical to their relational structure.

In particular, two concepts may be closely associated but not have a well defined relation: "sand" and "shells" for instance. Similarly, two concepts may be related, but not closely associated, as most people know abstractly but rarely use the fact that the "sun" is a "star."

Contexts are a natural way to organize memory, almost by definition: if several items in a context are in memory, then making other ones quickly accessible is a good idea because items in the same context are frequently used together. Organization of contexts into hierarchies might result in further benefits. Perhaps for very large scale systems, maintaining a memory hierarchy similar to computer hardware memory hierarchies but based on a context hierarchy would work. Suffice to say, there is much work to be done in this area and indexing and retrieval issues will be increasingly important for larger and larger scale systems.

Learning agent utilities and probabilities

Within the current specification, if there are multiple applicable agents, one is selected based on its utility. Unfortunately, presently utilities must be hard-coded within this system. It should be mentioned that many other proposals for conflict-resolution, such as choosing the most specific rule (Cawsey, p. 32), have been proposed for production systems, but utilities are perhaps the most general and flexible approach. Utilities can be estimated based on the time and quality of task completion. Choosing the maximum utility strategy may also not be the best approach: there is the classical exploration vs. exploitation tradeoff researched in the area of reinforcement learning (Mitchell, p. 369).

An agent need not have a single utility value either. General-purpose agents can have different utilities estimated for the different subclasses of problems to which they are applicable.

Moreover, the subtask decomposition provides for a natural way to transfer learning across tasks. In particular, we propose that the system estimate utilities of agents performing subtasks based only on their successful completion of their subtasks (and not on the ultimate success of the tasks of their superior agents). Then suppose that tasks A and B share a common subtask S. If the system performs task A many times, and learns that agent M is good at doing S, then if it is given task B, it will choose agent M to perform task S. This is similar to human skill transfer, which occurs when tasks are perceived as the same (Anderson, p. 307).

A similar type of learning that could easily be implemented with the system is the estimation of the proportions of members in each subset of a set (the probabilities). For instance, the system might estimate the distribution of ages of people it encounters or the proportion of mathematical functions encountered in its experience that are polynomials. Of course, it may be told contradictory proportions actually exist in the world. Both may be useful for different purposes.

Expanded declarative expressiveness, representation of beliefs

As the evaluation on the breakfast problems shows, the biggest limitations of the present specification are its difficulty in representing complex declarative assertions (e.g. "Some people eat five small meals a day") and abstract ideas like beliefs of other agents. There are multiple ways the system could be extended to increase its declarative

expressiveness. For representing beliefs, in particular, though, we strongly oppose the syntactic theory of mental objects (Russell & Norvig, p. 342), which can only be described as a hack.

Our approach is much more inspired by cognitive psychology and cognitive development. We maintain that the simplest statement an agent should be able to make about the minds of other agents is that "they know what I know." Then it can proceed to model particular differences between other minds and its own and to track the changes in other minds based on observation.

Natural language processing

Classical approaches to natural language processing have attempted to represent the semantics of English text streams in first-order logic and other knowledge representations (Russell & Norvig, p. 810). After this language has been implemented, it would be natural to attempt to communicate with it through English. Ultimately, this is the goal.

Development and meta-cognition

These more subtle types of knowledge were omitted from the initial set of breakfast benchmark problems. Work on development can investigate the ways in which the bias of the representation language can be changed. In particular, one could make the language modifiable by the user, so that the user can introduce new syntactic structures as convenient. The area of meta-cognition, through which an agent can monitor and improve itself, also offers great potential in the long run.

Acknowledgments

We would like to thank Prof. Tom Mitchell for his guidance, patience, support, and inspiration throughout this year.

References

Anderson, J., Bothel, D., Byrne, M.D., Douglass, S., Lebiere, C., and Qin, Y. "An Integrated Theory of the Mind." Available online at: http://act-r.psy.cmu.edu/tutorials/ 2004.

Anderson, J. *Cognitive Psychology and Its Implications*, Worth Publishers 2000.

Brachman, R., and Levesque, H. *Knowledge Representation and Reasoning*, Elsevier 2004.

Cawsey, A. *The Essence of Artificial Intelligence*, Prentice Hall Europe 1998.

Flavell, J., Miller, P., and Miller S. *Cognitive Development*, Prentice Hall 2002.

Harrison, R. *Universal Language Dictionary*. Available online at http://www.rick.harrison.net/uld/uld2.html 1996.

Lakoff, G. *Conceptual Metaphor System*. Available online at http://cogsci.berkeley.edu/lakoff/MetaphorHome.html 1994.

Larson, L. *Problem-Solving Through Problems*, Springer-Verlag 1983.

Lehman, J.F., Laird, J.E., & Rosenbloom, P.S. "A gentle introduction to Soar, an architecture for human cognition," In S. Sternberg & D. Scarborough (Eds.) *Invitation to Cognitive Science* (Volume 4), 1996.

Lenat, D. "From *2001* to 2001: Common Sense and the Mind of HAL." Available online at http://www.cyc.com/cyc/technology/halslegacy.html 2001.

Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, Prentice Hall PTR 1988.

McCarthy, J. "From Here to Human-Level AI." Available online at http://www-formal.stanford.edu/jmc/human/human.html 1998.

McCarthy, J. "Programs with Common Sense." Available online at http://www-formal.stanford.edu/jmc/mcc59/mcc59.html 1959.

Minsky, M. *The Emotion Machine* (draft). Available online at http://web.media.mit.edu/~minsky/E6/eb6.html 2004.

Minsky, M. *The Society of Mind*, Simon & Schuster, Inc 1986.

Mitchell, T. *Machine Learning*, McGraw Hill 1997.

Pinker, S. *How the Mind Works*, W. W. Norton & Company 1997.

Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall 2003.

Turing, A. M. "Computing Machinery and Intelligence." Available online at http://www.abelard.org/turpap/turpap.htm 1950.

Willingham, D. *Cognition: The Thinking Animal*, Prentice Hall 2001.

Zeitz, P. *The Art and Craft of Problem Solving*, John Wiley & Sons 1999.

Appendix: The Breakfast Problems

Declarative knowledge

Groups

1) What are the major meals? Breakfast, lunch, dinner.

Relations

2) Some people eat five small meals a day.
3) Breakfast is usually smaller than dinner.
4) Brunch is a combination of breakfast and lunch. (Combination of foods, replaces breakfast and lunch in terms of meals.)

Change

5) When do the major meals occur? morning, noon, evening
6) If people are hungry and eat enough, they are no longer hungry.
7) If people are not hungry and they don't eat, they become hungry gradually.

Uncertainty

8) Which would someone most likely eat for breakfast: a donut, a sandwich, a steak? A donut.
9) Which meal would most likely involve a donut? Breakfast.

Higher-order

10) John believes that Mary ate cereal for breakfast yesterday. John is wrong. Did Mary eat cereal for breakfast yesterday?

Meta-knowledge

11) Episodic memory: what did you have for breakfast yesterday? Frosted flakes, orange juice, coffee, toast

Context

12) John's breakfast tomorrow will have toast.
13) In China, rice is common for breakfast.

Procedural knowledge

Problem solving

14) How many meals does a person eat in a year?
    -There are 365 days in a year.

-People eat 3 meals per day.
-So a person eats 365*3 meals in a year.

Reasoning

15) John is eating Frosted Flakes. What time of day is it? Probably morning.

Control

16) If you are thirsty, drink.

Retrieval

17) Given:
John ate Frosted Flakes for breakfast yesterday.
Answer efficiently:
What did John eat for breakfast yesterday? (Frosted Flakes, others?)
Who ate Frosted Flakes for breakfast yesterday? (John, others?)
For which meal did John eat Frosted Flakes yesterday? (breakfast, others?)
When did John eat Frosted Flakes for breakfast? (yesterday, others?)

Sequential procedures

18) Compute the calories of the following breakfast: 1 bowl of cereal (made of 1 cup Frosted Flakes and ½ cup milk), 1 cup orange juice.

Pattern recognition
pattern recognition, similarity, assimilation, analogies, defaults

19) Compare breakfast and lunch.
Breakfast comes before lunch and is usually smaller.
20) Based on the concept of brunch, what would "linner" be like? How about "dinnfast"?
Linner would combine lunch and dinner. It would include foods from both lunch and dinner, come between the normal lunch and dinner times, and replace them.
21) Analogy
thirsty : drink :: hungry : ? (eat)

Pattern generation

22) Describe a breakfast.
23) Describe a healthy breakfast.