# Overview of an Approach to Representation Design*

**Jeffrey Van Baalen** and **Randall Davis**
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
jvb@ht.ai.mit.edu

## Abstract

It has long been acknowledged that having a good representation is key in effective problem solving. But what is a "good" representation? We describe an approach to representation design for problem solving that answers this question for a class of problems called analytical reasoning problems. These problems are typically very difficult for general problem solvers, like theorem provers, to solve. Yet people solve them quite easily by designing a specialized representation for each problem and using it to aid the solution process. Our approach is motivated, in large part, by observations of the problem solving behavior of people.

The implementation based on this approach takes as input a straightforward predicate calculus translation of the problem, tries to gather any necessary additional information, decides what to represent and how, designs the representations, then creates and runs a LISP program that uses those representations to produce a solution. The specialized representation created is a structure whose syntax captures the semantics of the problem domain and whose behavior enforces those semantics.

## 1 Introduction

It has long been acknowledged that having a good representation is key in effective problem solving. But what is a "good" representation? Most answers fall back on a collection of somewhat vague phrases, including "make the important things explicit; expose natural constraints; be complete, concise, transparent; facilitate computation" [Winston84]. These are of some assistance, but leave unresolved at least two important issues. First, saying that a "good" representation makes the "important" things explicit really only relabels the phenomenon – How are we to know what is important? Second, while phrases like these can conceivably serve as recognizers, allowing us to determine whether a given representation is good, little progress has been made on understanding how to *design* a good representation prospectively.

We have developed a new approach to this problem with a number of interesting properties:

- It begins with the initial problem statement, assists in determining what is "important" and hence what to represent, then helps identify any missing information required to solve the problem.
- It offers a more technical explanation of what makes for a good representation, claiming that it one whose syntax "captures the semantics" of the problem domain and whose behavior enforces those semantics by maintaining invariants in the syntax.
- Our approach shows how to *design* a representation with these properties, then how to solve the problem using that representation.

A demonstration of the approach has been implemented and tested on a small number of verbal reasoning problems of the sort found on graduate school level admissions tests. One of the problems, shown in Figure 1, is used throughout the paper for illustration. Our system takes as input a straightforward predicate calculus translation of the problem, gathers any necessary additional information, decides what to represent and how, designs representations tailored to this specific problem, then creates and runs a LISP program that uses those representations to produce a solution.

> *Given:* M, N, O, P, Q, R, and S are all members of the same family. N is married to P. S is the grandchild of Q. O is the niece of M. The mother of S is the only sister of M. R is Q's only child. M has no brothers. N is the grandfather of O.
> *Problem:* Name the siblings of S.

Figure 1: An Analytical Reasoning Problem

We document how the system does this, describing how it decides to both define and represent concepts like COUPLE, CHILDREN-OF, and CHILD-SET, even though those do not appear in the problem statement. We illustrate how the LISP program it creates solves the problem efficiently because it has a good representation.

## 2 Motivation

Our approach is motivated in large part by observations of the problem solving behavior people exhibit when solving problems of the sort shown in Figure 1, and inspired by the striking difference between that behavior and what we might call a "classroom logic approach."

*This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the authors' artificial intelligence research is provided by Digital Equipment Corporation, Wang Corporation, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

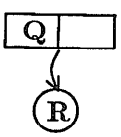
The classroom logic approach would begin by translating the problem into predicate calculus (Figure 2), then use a theorem prover to search for a solution.

| | |
|---|---|
| M, N, O, P, Q, R, and S are all members of the same family. | $M \in FAMILY, \ldots,$ $S \in FAMILY$ |
| N is married to P. | $married(N, P)$ |
| S is the grandchild of Q. | $grandchild(S, Q)$ |
| O is the niece of M. | $niece(O, M)$ |
| The mother of S is the only sister of M. | $mother(S, x) \Leftrightarrow sister(M, x)$ $[sister(M, x) \wedge sister(M, y)] \Rightarrow x = y$ |
| R is Q's only child. | $child(Q, x) \Leftrightarrow x = R$ |
| M has no brothers. | $\neg brother(M, x)$ |
| N is the grandfather of O. | $grandfather(O, N)$ |
| Name the siblings of S. | $find\text{–}all\ x(sibling(S, x))$ |

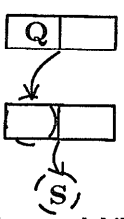
Figure 2: Translation to PC. (Upper case symbols are constants, lower case symbols used as arguments are universally quantified.)

One difficulty with this approach is that the problem specification (and hence its translation into PC) is incomplete: nothing in Figure 2, for instance, indicates that the relation *married* is symmetric. Once identified, that information is easily encoded as additional axioms; the harder part is knowing what is missing: on this task predicate calculus offers us little or no guidance.

More important from our perspective is that even a moderately experienced human problem solver would not proceed in this fashion, using an unstructured collection of axioms. He would instead *design and use specialized representations* and as a direct result produce solutions far more effectively. By a specialized representation we mean the sort of thing illustrated in Figure 3, which shows two of the sample problem statements in a representation people commonly use.




"R is the only child of Q"
Figure 3a.




"S is the grandchild of Q"
Figure 3b.

(Divided rectangles represent couples; circles represent sets of children of the same couple: full circles are closed sets, broken circles are sets all of whose members may not be known; the directed arc represents the "children-of" function between couples and their sets of children.)

Such representations are powerful because they capture the semantics of the problem domain, in two ways: (i) structurally: the structure of the representation resembles the structure of the thing represented (i.e, they are "direct" [Sloman71]), and (ii) behaviorally: associated with the structure are behaviors that are efficient in enforcing the semantics of the problem domain. We illustrate both of these informally here using the "children-of" link; a more formal discussion appears in Section 5.

In Figure 3 the "children-of" link captures in its structure the relation (a 1-1 function) between a couple and their set of children, because its syntax indicates that it is a pointer from one object (a couple) to one object (a set of children). Other aspects of the semantics are captured behaviorally: associated with the link are behaviors that reflect the fact that it is a function (and hence $x = y \Rightarrow f(x) = f(y)$): one behavior infers that two children sets are identical when they are the "children-of" the same couple. Because it is in addition a *1-1* function, another behavior can infer that two couples are identical if they are parents of the same children sets.

Inference is done in these representations by a composition process that is controlled by the behaviors. For example, consider what happens as the structures in 3a and 3b are combined. Using the fact that couples are disjoint, if we have what appears to be two distinct couples (the top box in Figure 3a and 3b) and also know that they share an individual (Q), then they are in fact the same and hence can be combined. Using a behavior that embodies this fact and the behaviors associated with "children-of," figures 3a and 3b can be combined to yield Figure 4, making clear that R is the parent of S.

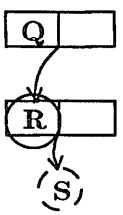



Figure 4: Composition of the Structures in Figure 3

This composition process is of fundamental importance because the representation that results from composing structures always contains *all* the deductive consequences of the conjunction of the composed statements. Problem solving using these representations involves composing the separate problem statements together into a single structure, then inspecting that structure for the solution. Composition is also a tightly constrained local process guaranteed to halt whether or not a solution exists.

The task of our system is to design representations like these, by picking out the important concepts in the problem (such as "couples" or "the siblings of an individual") and finding ways to operate on them using special purpose manipulations of the sort illustrated by Figure 4. Our system chooses what to represent and how, then solves the problem using those representations. In fact, it solves the problem by designing and using, among others, the representations illustrated in Figure 3.[1]

In the rest of this paper we follow this process through, using the concept of couple and children-of as key examples of the representation design process, and exploring the origin of the specialized inference rules illustrated by Figure 4.

[1] While our system expresses those representations in terms of data structures and procedures, language is not so much the issue: much the same effect can no doubt be accomplished by a skilled logician carefully selecting axioms, lemmas, and special purpose inference rules. Whatever the language, the important point is selecting carefully — the representations and inference knowledge are specialized to the problem — and capturing the semantics in the manner suggested above.

# 3 Terminology, Typography

The problem of representation design appears to consist of at least three different decisions: what to represent, how to represent it, and how to implement those representations. Determining what to represent involves deciding "what to pay attention to" — identifying the relevant domain concepts and properties. In the example problem of Figure 1, it is the decision to think of the problem in terms of couples, sets of children, etc. Next we have to decide how to represent those concepts and properties. Having decided to pay attention to couples, for instance, it is useful to determine that they form a partition,[2] since, as we have seen, this allows us to use a specialized inference rule. Third is the familiar choice of data structures: determining whether to implement a set as a list, array, bit vector, etc.

Our approach makes its contributions at the first and second levels; questions at the third level — data structure selection – have been studied elsewhere (e.g., [Barstow79]).

In the rest of the paper concepts found in the predicate calculus statement of the problem (Figure 2) are written using italics (e.g., *married*). The system has a library of types (described below), consisting of mathematical entities like `set`, `fixed-size-set`, and `equivalence-relation`, noted with a typewriter-style font. Those types are in turn used as building blocks to construct our representations, things like COUPLE, SIBLING-SET, and PARENTS, noted with a small-caps font.

# 4 Knowledge For Representation Design

An important foundation for our approach is a body of knowledge called the *type library* (Figure 5). The types are used as building blocks in designing specialized representations. Each type contains a data structure and its associated manipulation procedures. The `set` type, for instance, contains a list data structure to indicate one way of implementing a representation (like COUPLE) built from the `set` type and procedures for manipulating sets like "add element" and "test for equality."

The type library is organized as a pair of mathematical concept taxonomies, with `set` and `relation` as the two roots and specialization links labeling the additional properties that the more specialized types have. Those types have additional procedures associated with them that exploit their properties to provide additional functionality efficiently. A procedure associated with `partition-element`, for example, exploits disjointness to determine efficiently when two elements of a partition are the same.

Note that only some of the nodes in the concept taxonomy have a type label, reflecting the knowledge that those nodes are useful building blocks for representations. The system currently does not, for example, have a type for a binary, symmetric, intransitive relation.

[2] A partition is a set of disjoint sets.

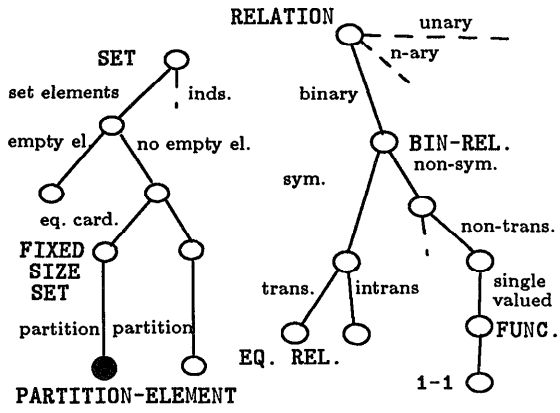



Figure 5: A Portion of the Type Library. The uppercase labels are names of types.

Another important knowledge source is a set of concept revision heuristics. These use properties of existing concepts in a representation or structural properties of a problem statement to revise concepts into a form that often proves to be more useful for problem solving. There are currently 12 such heuristics.

The heuristics that use properties of existing problem concepts are associated with nodes in the type library taxonomies. These use the properties of the node to which they are attached to suggest reformulations. Several examples of this type of heuristic are given below.

The other type of heuristics look for structural features in the problem statement. When a feature is found, the relevant heuristic suggests a revision. Consider, for example, the statement "M has no brothers," can naturally be (re)expressed as a constraint on the cardinality of the set of M's brothers. One heuristic embodies this intuition by looking for negation at the top level in universally quantified formulas. When it finds the problem statement about M's brothers, it revises "brother" to "brother-set."

Whenever a concept is revised, the problem statement is rewritten to reflect this change. The above revision, for instance, causes the problem to be rewritten in terms of "brother-set." As a result, the original formula is rewritten as $\{x \mid brother(M, x)\} = \emptyset$.[3]

# 5 Representation Design

The goal of representation design is to create a representation that "captures the semantics" of the original problem statement. Earlier we gave an informal definition, indicating that capturing semantics can be accomplished by attention to the structure and behavior of a representation: its structure should mimic the thing represented and its behavior should enforce the problem semantics. To be more precise, we say that a representation captures the semantics of a set of formulas when the possible data structures that can be built from it satisfy those formulas.[4]

[3] A condition placed on concept revision heuristics guarantees that the revisions preserve satisfiability by showing that any model of the original problem can be extended to include the new concepts.

[4] Technically, we define a new satisfaction relation between representations and sets of formulas as follows. We define a

Consider for example the formula:

$$\forall x \forall y[couple(\{x, y\}) \Leftrightarrow couple(\{y, x\})]$$

We say that the representation COUPLE captures the semantics of this formula because COUPLE is defined in terms of `set`, whose semantics indicate that the two sets $\{x, y\}$ and $\{y, x\}$ are equal. Thus, any instance of COUPLE will have the property that whenever $\{x, y\}$ is a couple, $\{y, x\}$ is a couple. A specialized representation is *complete* when all the formulas in the problem statement have their semantics captured.

## 5.1 The Process of Representation Design

Representation design is a three step process: representation introduction, dependent representation introduction, and operationalization. The first two of these are incremental processes aided by concept revision heuristics. In general, multiple concept revisions can occur in an effort to allow further representation introduction.

This helps to illustrate two important aspects of our approach to the problem. First, we believe that good representation design is fundamentally an incremental, opportunistic process that proceeds best in small steps with constant rewriting of the problem as the representation evolves. Second, as we will see in exploring the use of the type library, we believe the process should be informed and guided by both the problem statement and the set of representations available.

Since representation introduction does not in general produce a *complete* specialized representation, the goal of *operationalizing* is to generate procedures that extend the representations, to ensure that their behavior captures the semantics of the remaining formulas.

In the remainder of this section we work through what the system does in designing a representation for the problem in Figure 2. The input to our system is a straightforward translation of the problem statement into predicate calculus, i.e., exactly the set of formulas shown in Figure 2. The sequence of actions explored below is the first successful path completed by the system when given the example problem; there are roughly a half dozen other paths explored but left uncompleted because the system halts with the first successful one.

## 5.2 Representation Introduction

The system begins by attempting to find types in the library that will prove useful in designing representations for concepts mentioned in the problem. This is in turn an iterative process of *taxonomic classification* and concept revision.

Taxonomic classification is performed on each primitive relation and each set in the problem statement, using the specialization links to decide what properties to investigate. Consider, for example, the relation *married*: Figure 5 indicates that relations are specialized first in terms of degree. The system is able to determine by inspection that *married* is binary. Following the links down, the system encounters the issue of symmetry, then transitivity; *married* is symmetric and intransitive, at which point we arrived at a leaf node.

Since there is no type at this node, the system checks to see if there are any concept revision heuristics associated with the node that can suggest ways to revise the current concept. One such heuristic suggests restating the relation *married* in terms of sets of individuals married to a fixed individual, i.e., replace assertions of the form $married(x, y)$ with sets of the form $\{x \mid married(p_1, x)\}$ (where $p_1$ is an arbitrary individual). This introduces a new concept, the set of all sets of this sort (call it *set–of–spouse–sets*), and completes one classification/revision cycle.

A second cycle begins with another classification effort, this time at the `set` node trying to specialize *set–of–spouse–sets*. Following the taxonomy, the system determines whether the elements of this set are themselves sets (yes) and then whether $\emptyset$ is an element (yes, since not all people are married). Once again classification ends and a revision heuristic at this node suggests, "if a set S contains $\emptyset$, try introducing the set equal to $S - \emptyset$." This is accomplished by restricting x to be a married individual in $\{y \mid married(x, y)\}$.

A third classification effort now begins at the `set` node. It determines that the new concept is a set of sets not containing $\emptyset$ and that all the element sets have cardinality 1 (it has now reached the `fixed-size-set` node). Again classification halts and a revision heuristic found that states, "when sets of the form $\{y \mid R(x, y)\}$ all have cardinality 1 and R is symmetric, introduce sets of the form $\{y \mid R^*(x, y)\}$ where $R^*$ is the equivalence relation defined by $\forall x \forall y[R^*(x, y) \Leftrightarrow x = y \vee R(x, y)]$." [5]

This introduces the set of sets of the form $\{y \mid married^*(x, y)\}$, where $x$ is restricted to married individuals. Each element of this set is a set of married people, i.e., our notion of a couple (call this set *couples*).

Once again this concept is classified; it is a set of sets, it does not contain the empty set, each of the member sets is of the same size (cardinality), and it is a partition. Hence the process arrives at the shaded node in Figure 5. This time the process halts, because it has arrived at a node that does have an associated type (`partition-element`) and does not have any revision heuristics.

This intertwined process of classification and concept revision has thus in several cycles transformed the problem from one using assertions phrased in terms of the *married* relation to one phrased in terms of *couples*. As noted, when new concepts are introduced, the problem is rewritten. When *couples* is introduced, for example, all formulas using the term $married(x, y)$ are rewritten to use it instead. The formula $married(N, P)$ in the original problem statement, for instance, is rewritten to $couple(\{N, P\})$ (shorthand for $\{N, P\} \in couples$).

As a result of this process, a representation is introduced by providing a definition of an abstract data type

class of functions from data structures to logical models. A representation satisfies a set of formulas just in case there is a function from this class mapping the data structure built by that representation to a model of the formulas.

[5] The idea here is that we would like to find a partition because there is a powerful specialized inference procedure associated with it. One way to identify a partition is to look for an equivalence relation that induces it; this heuristic suggests one way to define such a relation.

and establishing a correspondence between that data type and a symbol in the problem statement. The representation COUPLE, for example, is introduced by defining it in terms of the library types for `partition-element` and all of its ancestors (i.e., `set` and `fixed-size-set`), then establishing a correspondence between COUPLE and the symbol *couple*. (Henceforth we will refer to *couple* as a "represented symbol" because there is a representation associated with it.)

### 5.3 Dependent Representation Introduction

The goal of this process is to introduce additional representations that are connected to representations introduced in the previous process, relying here on heuristics that look for structural features in the problem statement. These heuristics have an additional constraint, they look for features in the context of existing representations.

To see an example, first recall that each time a new representation is introduced the problem is rewritten in terms of it. One formula that appears in the problem statement when *couple* is introduced is:

$$\forall p_1 \forall p_2 \exists c[child(p_1, c) \wedge child(p_2, c) \wedge p_1 \neq p_2 \Rightarrow married(p_1, p_2)]$$

(i.e., the parents of an individual are married).[6] This formula gets rewritten in two ways. It uses the term *married*, so it will be rewritten when *couple* is introduced. It also uses the relation *child*; during taxonomic classification of this relation the system discovers that the set $\{x \mid child(x, c)\}$ (i.e., the parents of an individual) has cardinality 2. A concept revision heuristic at the node for asymmetric intransitive relation will introduce this set as a new concept and the formula above will be rewritten in terms of it. The final result of these (and other) transformations is:

$$\forall c[couple(\{x \mid child(x, c)\})]$$

One interesting thing about the revised formula is that it makes clear that the newly created set is related to an existing representation: $\{x \mid child(x, c)\}$ in fact *is* a COUPLE (i.e., the parents of a child are a couple). The following concepts revision heuristic now becomes applicable: "any set of the form $\{y \mid R(x, y)\}$ appearing as an argument to a represented symbol (in this case *couple*) should be viewed as a function $F(x) = \{y \mid R(x, y)\}$." A new concept is introduced, a function we will call *parents*, mapping individuals to *couples*.

When *parents* is introduced, the formula above is rewritten as $\forall c[couple(parents(c))]$. Another concept revision heuristic indicates that "from any many-to-1 function F whose range elements correspond to a representation, create a 1–1 function: create the sets $\{x \mid F(x) = y\}$ (i.e., all the domain elements that map to the same range element); then create the 1–1 function $G : \{x \mid F(x) = y\} \rightarrow y$."

Invoking this heuristic means introducing two new representations: CHILD-SET for sets of the form $\{y \mid x = parents(y)\}$ and CHILDREN-OF, a one-to-one function from COUPLEs to CHILD-SETs.

[6] The system acquires this formula while identifying missing information, a process not fully described here. For the current purposes, assume the formula was given.

While it does not occur in this example, the concept revisions that occur in this phase can cause new classification efforts to begin, sending us back to the representation introduction phase.

Note finally that at this point we have created much of the specialized representation shown in Figures 3–4, i.e., COUPLEs, PARENTS, and CHILD-SETs. The process that yields the structure in Figure 4 from the two structures in Figure 3 is realized by the procedures associated with the `partition-element` and `1-1 function` types.

### 5.4 Operationalization

As noted, representation introduction captures much, but not all, of the semantics of the problem statements by selecting appropriate types from the library. Operationalizing is a way of extending representations so that they capture the semantics of problem statements not already captured.

A formula is operational when it can be interpreted as code built from just the operations associated with the types chosen from the library. For example, suppose that SIBLING-SET is a representation defined as a `set` of individuals, and that SIBLINGS is a function from an individual to that individual's sibling-set. Then

$$\forall x \forall y[x \in siblings(y) \Leftrightarrow y \in siblings(x)]$$

is operational because: SIBLINGS is defined as a `set`, one of the operations associated with the type `set` is add-element, and we can interpret the entire formula as a (demon-like) procedure using only the available operations (in this case, just add-element): `whenever x is added to the siblings(y), add y to the siblings(x)`.

We claim this procedure "captures the semantics" of the formula above because it ensures that the data structures used to implement SIBLING-SET will, at execution time, maintain the relationship expressed by the formula. That is, it executes when a representation is modified in the process of solving the problem in Figure 2, and responds by making corresponding changes to other representations. Making a formula operational captures the semantics by making the formula itself do the work: we turn a statement of the relation into a procedure that enforces the relation.

The process of operationalizing formulas not already in that form is rather complex and lengthy, details are in [Van Baalen88].

## 6 Solution

The solution phase uses the representations and procedures to solve the original problem. Our system translates the output of the previous two phases into an object oriented LISP program, which is then executed to solve the problem. For the problem of Figure 2, the specialized representation is derived in about twenty minutes on a Symbolics 3600; the corresponding LISP program is created in about five minutes; the LISP program in turn requires less than five seconds to produce the correct answer that O is the only sibling of S.

Recall that the important task of the previous phases was to capture the semantics of the formulas in the problem statement, and that this was done by (i) defining appropriate representations (like COUPLE), and (ii) putting the remaining formulas into operational form. The translation

to LISP is then achieved simply by (i) translating the representations into LISP flavor definitions, and (ii) translating the operational formulas into methods of the appropriate flavors. Again details are in [Van Baalen88].

# 7 Related Work

Problem reformulation has a long history (e.g., [Newell66]); a recent effort in this vein, [Korf80], is the most direct ancestor of ours. Our theory extends this work: it begins with an incomplete problem and, among other things, identifies relevant operations in the type library.

[Bobrow68]'s STUDENT program solved algebra word problems and is similar in going from a problem specification to creation of a representation and solution. The problems it solved, however, are much simpler, are not missing information, and most important, are stated in a vocabulary that is appropriate for their solution.

Efforts to understand the notion of direct or analogical representation define it in terms of a representation syntax reflecting the problem semantics([Sloman71], [Hayes74], [Pylyshyn75]). [Pylyshyn75] points out that one can speak of representation structure only with respect to a Semantic Interpretation Function, by which he means "the processes which construct and use the representation." The representations we design come complete with procedures that construct and interpret them. Furthermore, we specify how to understand the meaning of structures built in our representations in terms of formal models.

# 8 Comments and Summary

Our approach to representation design is based on two claims. First, in the context of a problem, we should design a representation whose syntax captures the semantics of the problem domain and whose behavior enforces those semantics by maintaining invariants in the syntax. Second, it tells us what knowledge to use in specializing representations and how this knowledge should be organized. The type library, for instance, defines a collection of types used to define a specialized representation and is organized as a taxonomy to facilitate finding maximally specific data types. The library also organizes semantic reformulation heuristics.

Our approach differs from more traditional approaches to problem solving because it begins at an earlier stage of the problem solving process, starting with the problem statement and identifying missing information required to solve the problem, and because it explains how to find a more useful problem solving vocabulary and how to design specialized representations from it.

The approach is also still in an early form and as such has some weaknesses. The representation design heuristics, for example, while based on broadly applicable mathematical principles, are still somewhat ad hoc and the intuitions underlying them are not always clear. We continue to look for a more methodical underpinning to them.

We also have only a preliminary characterization of the class of problems for which the current knowledge bases in the implementation are applicable. This is the class of problems that can be effectively solved by identifying and reasoning about extensional sets. We continue to look for a more precise characterization. Also we believe the general approach applies to a far wider class of problems.

There is also an implicit claim in using the type taxonomy to drive information gathering: it assumes that asking about the properties we know how to exploit in problem solving will be effective in determining what properties of a domain concept will be needed to solve the problem and what properties will be useful in solving the problem. Both of these are clearly only sometimes true and depend on the size and sophistication of the library: taxonomic classification may fail to enquire about properties that are in fact necessary to solve the problem (in which case the problem simply won't be solved), and may fail to gather facts that would have been useful. This latter phenomenon is less serious because of the ability of operationalizing to capture semantics that the type library misses.

Despite its early stage of development, our approach made it possible for our program to start with a simple predicate calculus translation of a problem, gather necessary information, decide what to represent and how, design representations, create a LISP program that uses those representations, and finally run the program to produce the solution. It has succeeded in doing this for a small number of quite different analytical reasoning problems.

**Acknowledgments**

Useful comments on drafts of the paper were received from Yishai Feldman, Walter Hamscher, Reid Simmons, Dan Weld, Brian Williams, Patrick Winston, and Peng Wu.

# References

[Amarel68] Amarel, S., "On Representations of Problems of Reasoning About Actions," In Michie, D. (editor), *Machine Intelligence 3*, pp. 131-171, Edinburgh University Press, 1968.

[Barstow79] Barstow, D., "An Experiment in Knowledge-Based Automatic Programming," *Artificial Intelligence,* 12, pp.73-119, 1979.

[Bobrow68] Bobrow, D.G., "Natural Language Input for a Computer Problem-Solving System," in Minsky, M. (editor), *Semantic Information Processing*, pp.146-226, MIT Press, 1968.

[Hayes74] Hayes, P.J., "Some Problems and Non-Problems in Representation Theory," in Brachman, R.J., and Levesque, H.J. (editors), *Readings in Knowledge Representation*, pp.3-22, Morgan Kaufmann, 1985.

[Korf80] Korf, R.E., "Toward a Model of Representation Changes," *Artificial Intelligence,* 14, pp.41-78, 1980.

[Newell66] Newell, A., "On Representations of Problems," in *Annual research review,* Department of Computer Science, Carnegie-Mellon University, 1966.

[Pylyshyn75] Pylyshyn, Z.W., "Do we need images and analogs?" pp.174-177, TINLAP-1, 1975.

[Sloman71] Sloman, A., "Afterthoughts on Analogical Representations," in Brachman, R.J., and Levesque, H.J. (editors), *Readings in Knowledge Representation*, pp. 431-440, Morgan Kaufmann, 1985.

[Van Baalen88] Van Baalen, J., "A Theory of Representation Design," Ph.D. Thesis, forthcoming.

[Winston84] Winston, P.H.,*Artificial Intelligence*, Addison-Wesley Publishing Co., 1984.