Technical Report 1128

Toward a Theory of Representation Design

Jeffrey Van Baalen

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

Toward A Theory of Representation Design

by

Jeffrey Van Baalen
B.S., University of Wyoming, 1978
M.S., University of Wyoming, 1980

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

at the

Massachusetts Institute of Technology

January, 1989

©Jeffrey Van Baalen, 1989. All rights reserved. The author hereby grants MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

This empty page was substituted for a blank page in the original document.

Toward A Theory of Representation Design

Jeffrey Van Baalen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at the
Massachusetts Institute of Technology, January, 1989

Abstract

This research is concerned with designing representations for analytical reasoning problems (of the sort found on the GRE and LSAT). These problems are intended to test the ability to draw logical conclusions from information presented and to synthesize that information in order to deduce interrelationships.

A computer program was developed that takes as input a straightforward predicate calculus translation of a problem, requests additional information if necessary, decides what to represent and how, designs representations that capture the constraints of the problem, and finally, creates and executes a LISP program that uses those representations to produce a solution. Even though these problems are typically difficult for theorem provers to solve, the LISP program that uses the designed representations is very efficient.

The representations designed by the system are powerful because they capture the constraints of a problem in two ways. (i) The structure of the representation resembles the structure of the thing represented. For example, consider representing married couples as sets of size two. The structure of this representation resembles the structure of married couples because both have exactly two individuals in them. (ii) The structure enables efficient behaviors that enforce a problem's constraints by keeping them invariant in the structure. For example, as a set representing a married couple is manipulated, a behavior associated with it maintains its fixed size. This behavior efficiently enforces the size constraint on married couples.

The system designs a representation that captures as many of the constraints of a problem as possible. When a representation captures more constraints fewer sets of facts can be expressed in it. For example, in a representation that does not capture the symmetry of the married relation, one can state "A is married to B" and "B is not married to A." However, it is not possible to express these two statements in a

representation that captures symmetry. Allowing fewer sets of facts to be stated in a representation reduces the space that a problem solver must consider; this in turn results in more efficient problem solving behavior.

Representation design consists of three processes: *classification*, *concept introduction*, and *operationalization*. Classification uses a library of structures each capturing different kinds of constraints. It finds the most specialized library structure for representing each concept in a problem. Concept introduction is a way of enhancing classification. When classification fails to capture all the constraints on a concept, introduction tries different ways to represent the concept. For example, sometimes when classification fails to capture all the constraints on a relation like "married," it introduces a concept like "couple," a function mapping an individual to the set of individuals he/she is married to. Classification of "couple" results in a representation that captures more constraints than did the representation of "married."

Classification and concept introduction run as coroutines, trying to capture all of the constraints of a problem. As they do this, the statements of captured constraints get removed from the problem. However, in many problems these processes fail to capture all of the constraints of a problem, leaving statements of the uncaptured constraints. Operationalization then tries to capture the constraints of the remaining statements by writing procedures and using these to further specialize the representations created by the previous processes.

The demonstration system has designed representations for twelve analytical reasoning problems. In each case, designing a representation and using it to solve the problem has proven to be far more efficient than using a binary resolution theorem prover to search for a solution in the initial representation.

Acknowledgments

I would like to thank Randy Davis, a great and patient advisor.

I would like to thank Chuck Rich for his never ending help and support; Peter Szolovitz and Patrick Winston for their support.

I would like to thank my colleagues: Yishai Feldman, Walter Hamscher, David Kirsh, David McAllester, Mark Shirley, Howie Shrobe, Daniel Weld, Brian Williams, Peng Wu.

I would like to thank Lynne Staub, Justin Van Baalen, and Gwedolyn Van Baalen for their support and understanding.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part 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.

This empty page was substituted for a blank page in the original document.

Contents

1 Introduction **9**

1.1 Motivation 10

1.2 Overview of Representation Design 14

1.2.1 Representation 15

1.2.2 A Model of Representation Design 18

1.2.3 Representations and Problem Classes 20

1.2.4 Implementation of Representations 22

1.2.5 Knowledge about Representations 23

1.2.6 The Goal of Representation Design 24

1.2.7 Deriving an Initial Representation 25

1.2.8 Classification 26

1.2.9 Concept Introduction 30

1.2.10 Operationalization 34

1.3 Analytical Reasoning Problems 35

1.4 Soundness of Representation Design 39

1.5 Scope and Limitations 40

1.5.1 Coverage 40

1.5.2 Scope of Applicability 42

1.6 Related Work 43

1.6.1 Solving Word Problems 43

1.6.2 Automatic Programming 44

1.6.3 Good Representation 44
1.6.4 Problem Reformulation 44
1.6.5 Specialized Reasoners 45
1.7 Reader's Guide 45

2 Descriptions Used in Representation Design **47**
2.1 The Problem Statement Language 47
2.2 A Language for Defining Representations 48
2.3 Maintaining the Representation Mapping 51

3 Deriving an Initial Representation **55**
3.1 Identifying The Primitive Concepts of a Problem 58
3.1.1 Section Summary 59
3.2 Eliminating Irrelevant Information 60
3.2.1 Strong and Weak Irrelevance 60
3.2.2 An Approximate Strong Irrelevance Filter 61
3.2.3 Using the Connection Graph To Identify Definitions 70
3.2.4 Removing Definitional Irrelevance 71
3.2.5 Eliminating Irrelevance in Incomplete Problem Statements . . 72
3.2.6 Section Summary 73
3.3 Including Concepts With Restrictions 73
3.4 Completing a Problem's Sort Signature 74
3.4.1 Section Summary 76
3.5 Chapter Summary 76

4 Classification **78**
4.1 The Classification Process 83
4.1.1 Answering Questions Posed by Classification 86
4.1.2 An Example of Classification 88
4.2 Assumptions About Library Structures 91

5.6.5 Summary of Extended Classification of Child 141
5.7 Deriving New Mixed Constraints 142
5.8 Detecting Redundant Introductions 145
5.9 Chapter Summary 146

6 Operationalization **148**
6.1 Preprocessing for Operationalization 150
6.1.1 Removing Existential Quantifiers 151
6.1.2 Expanding Definitions 152
6.1.3 Removing Embedded Function Terms 152
6.2 The Operationalization Procedure 153
6.2.1 Identifying Operational Literals 155
6.2.2 Operationalization Sequences 156
6.2.3 Generating Procedures from Statements in Operational Form . 164
6.3 Soundness of Operationalization 169
6.4 Summary 171

7 Representation Machinery **174**
7.1 Underlying Mechanisms 175
7.1.1 The Equality System and Anonymous Individuals 175
7.1.2 The Daemon Invocation Mechanism 176
7.2 Implementation of Library ADTs 178
7.3 Solving the Example Problem 182

8 Related Work **186**
8.1 Solving Word Problems 186
8.2 Relationship to Automatic Programming 188
8.3 Research on Good Representation 192
8.4 Problem Reformulation 192
8.5 Psychology and Mental Models 196

9 Summary and Future Work **197**

9.1 Summary 197

9.1.1 Summary of An Example of Representation Design 201

9.2 Future Work 205

9.2.1 Experimental Work 205

9.2.2 Extending the Current System's Functionality 207

9.2.3 Representing Disjunction 208

A The Rewriting System **211**

A.1 Statement Simplification 212

A.1.1 The Current Collection of Simplifying Rewrite Rules 213

A.2 Goal Directed Derivation 214

This empty page was substituted for a blank page in the original document.

Chapter 1

Introduction

It has long been acknowledged that good representations are key in effective problem solving. But what is a "good" representation? Most answers fall back on a collection of somewhat vague phrases, such as "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 of good representations, little progress has been made on understanding how to *design* a good representation prospectively.

I have developed a new approach to designing good representations. The approach has the following key features:

- It begins with an initial problem statement. It determines what to represent and assists in identifying missing information that is required to solve the problem. Thus it assists in determining what is "important" in a problem statement.
- It offers a technical explanation of what makes for a good representation, claiming that it is one that captures constraints of a problem *directly* in its structure and behavior rather than leaving the constraints to be enforced by the problem solver using that representation.
- It shows how to *design* a good representation, and then how to solve the problem

using that representation.

I have implemented a demonstration of this approach, which I call the *representation design system*, and have tested it on a small number of verbal reasoning problems of the sort found on graduate level standardized admissions tests. One of these problems, shown in Figure 1.1, is used in this thesis for illustration. This problem will be referred to as the FAMILIES problem.

The system takes as input a straightforward predicate calculus translation of a problem, requests additional information if necessary, decides what to represent and how, designs representations tailored to the problem, and finally creates and executes 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.
> *Query:* Who are the siblings of S.

Figure 1.1: The FAMILIES Analytical Reasoning Problem

1.1 Motivation

My approach is motivated in large part by my own observations of the problem solving behavior people exhibit when solving problems of the sort shown in Figure 1.1, and inspired by the striking difference between that behavior and what we might call the "classroom theorem-proving approach." This approach begins by translating the problem into predicate calculus (Figure 1.2). Then a general purpose theorem prover (e.g., using binary resolution) is used to search for a solution using the initial problem statements.

One problem with this approach is that the initial problem specification is incomplete: nothing in Figure 1.2, for instance, indicates that the *married* relation is symmetric or that *grandfather* is the father of the father, etc. Once identified, that information is easily encoded as additional axioms. The harder part is knowing what is missing: on this task the classroom theorem-proving approach offers us little or no guidance.

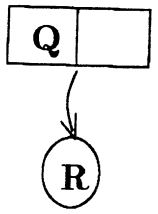
Given:	
M, N, O, P, Q, R, and S are all members of the same family.	$M : family\text{-}member,$ $\ldots, S : family\text{-}member$
N is married to P.	$married(N, P)$
S is the grandchild of Q.	$grandchild(S, Q)$
O is the niece of M.	$niece(O, M)$
The mother of S is the only sister of M.	$mother(S, x) \Leftrightarrow sister(M, x)$ $[sister(M, x) \wedge sister(M, y)]$ $\Rightarrow x = y$
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.	$grandfather(O, N)$
Query:	
Who are the siblings of S.	find-all $x \mid sibling(S, x)$

Figure 1.2: Line-by-line translation of FAMILIES problem to sorted first order logic. (Upper case symbols are zeroary constants. Lower case symbols in variable positions are universally quantified. Also note that we include a separate language (to be described later) for expressing a problem's queries. The above statement involving find-all is an example of a statement in this language.)

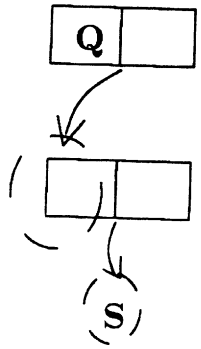
More important from the perspective taken in this research is that a human problem solver would not use an unstructured collection of axioms, but would instead *design and use representations that capture the constraints of the problem* to produce solutions far more effectively. An example of this type of representation is illustrated in Figure 1.3, which shows two statements of the FAMILIES problem in a representation people commonly use.

Such representations are powerful because they capture the constraints of a problem, in two ways: (i) the structure of the representation resembles the structure of the thing represented (i.e, they are "direct" [Sloman71]), and (ii) this structure enables efficient behaviors that enforce a problem's constraints by keeping them invariant in the structure. These are both illustrated here with the example "children-of" link.

The "children-of" link syntactically captures the 1-1 function between a couple and their set of children. There are also specialized behaviors associated with this link because it is a 1-1 function. One behavior uses the fact that "children-of" is a function to combine children sets when they are the "children-of" the same couple (i.e., if two couples are equal, the objects their "children-of" links point to are equal). Because



"R is the only child of Q"



"S is the grandchild of Q"

Figure 1.3: Two statements in a representation commonly used by people.
A divided rectangle represents a couple; a circle represents a set of children of the same couple: solid circles are closed sets, dashed circles are sets all of whose members may not be known; the directed arc represents the "children-of" function between a couple and their set of children.

it is a *1-1* function, another behavior does the same to couples that are parents of the same children sets.

Using the fact that couples are disjoint, if we have what appears to be two distinct couples (the top boxes in Figure 1.3) 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," the two structures in Figure 1.3 can be combined to yield Figure 1.4.

As will be detailed later, this combination process is of fundamental importance because it computes all the relevant[1] ground consequences of the conjunction of statements as they are combined. This process is tightly constrained by the syntax of the representations. For instance, Figure 1.4 represents all the relevant consequences of the conjunction of the structures in Figure 1.3 (e.g., "S is the child of R"). The consequences are computed by the behaviors described above performing local operations on the two structures.

The representation design system accepts a problem stated in predicate calculus as input. For example, the system is given the predicate calculus version of the FAMILIES problem (i.e., those statements shown on the right in Figure 1.2). The task of the system is to design representations like Figure 1.4, by picking out the important concepts in the problem (such as "couples" or "the siblings of an individual"), and

[1] Relevance will later be defined in terms of the problem query.

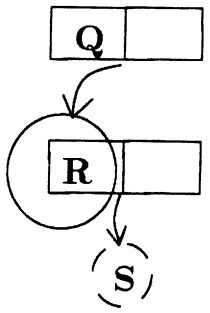


Figure 1.4: Composition of the structures in Figure 1.3.

finding ways to operate on them using special purpose manipulations of the sort illustrated by the transformation from Figure 1.3 to Figure 1.4. The system chooses what to represent and how, then solves this problem using those representations.

The demonstration system has been tested on the FAMILIES problem, on two other problems shown in Figure 1.5 and Figure 1.6, and on three variations of each. The system designs representations for all twelve problems. In each case, designing a representation and using it to solve the problem has proven to be far more efficient than using a resolution theorem prover to search for a solution in the initial representation. For example, representation design for the FAMILIES problem takes twenty minutes.[2] A LISP program using the representation is generated in five minutes; it solves the problem in less than three seconds. By contrast, missing information was added to the problem statement and then a connection graph resolution theorem prover was used to solve it. This took 988,442 resolutions — three hours and five minutes — to find a solution.

The representations produced by the system are expressed in terms of abstract data types: the data structure of the data type is used to implement the structure part of a representation; the procedures associated with the data type are used to implement the behaviors. For example, the system designs an ADT for couples in the FAMILIES problem. Instances of this type serve roughly the same function as the rectangles in Figure 1.3 and Figure 1.4: they represent specific couples like the one containing Q. The data structure part of a couple contains two slots for holding the two individuals in a couple. There are several procedures associated with couple. One procedure combines distinct couples if they have a common member.

[2]All times given are for programs running on a Symbolics 3600.

Eight law professors are housed in a single wing of a building. The wing contains ten offices, numbered 1 to 10, in that order; each professor is assigned to a different office, and two offices are left empty for use as meeting rooms. The professors are named Boswell, Dyer, Garrett, Harrelson, Kranepool, Ryan, Taylor, and Weis.
Dyer is four offices away from Kranepool.
There is one empty office and one occupied office between Taylor and Harrelson.
Ryan is in an office next to Boswell.
Dyer is in an office next to Garrett.
Kranepool is between an occupied office and an empty office.
Weis is in office 2.
Garrett is in office 7.
Who is in office 4?
Which offices are unoccupied?
Is Ryan, Dyer, Garrett, Taylor a possible sequence of offices?

Figure 1.5: The PROFESSORS problem

Note that while my system designs representations in terms of abstract data types, the language used to express the design is not so much the issue: much the same effect can no doubt be accomplished by a skilled logic programmer carefully selecting axioms, lemmas, and special purpose inference rules. Whatever the language, the important point is the careful selection of representations that capture the constraints of the problem.

1.2 Overview of Representation Design

This section provides an overview of the process of representation design. Chapter 2 discusses the descriptions that the system uses. The four subsequent chapters describe in detail the different elements of the representation design process.

Representation design begins with a *problem statement*, a collection of statements in a sorted first order logic, along with one or more queries written in a separate query language described later. The constant symbols in the statements refer to what we will call *concepts*: individuals, relations, and functions. For example, in the FAMILIES problem the statement,

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)],$$

the symbol *married* refers to the concept "married."

A restaurant employs eight waiters, D, E, F, G, H, I, J, and K, each of whom works four days a week. The restaurant is open every day except Monday. On Friday and Saturday, a staff of six waiters is needed. On all other days when the restaurant is open, a staff of five is needed.
D cannot work on Tuesday or Thursday.
E cannot work on Wednesday.
G cannot work on Thursday or Saturday.
H cannot work on Friday.
J cannot work on Tuesday or Sunday.
K cannot work on Wednesday or Friday.

Is D,E,F,I,K a possible staff of waiters for a Tuesday?
Is Tuesday, Wednesday, Thursday, Sunday a possible work week for G?

Figure 1.6: The WAITERS problem

A familiar notion in logic is that statements constrain the possible models of a problem. For example, the above statement constrains all models of the FAMILIES problem to be models in which "married" is symmetric. An important part of my method is to design representations that are constrained in the same way as the models of a problem.

1.2.1 Representation

A *representation* is a mapping between concepts and syntactic structures in a representation language. For instance, the representation of a problem expressed in first order logic is the mapping between the concepts mentioned in the problem statement and syntactic structures of first order logic. In this case, there are three syntactic structures: constants (i.e., symbols that appear as terms), relations (symbols followed by lists of arguments that appear as atomic formulas), and functions (symbols followed by lists of arguments that appear as terms). The syntactic category of a symbol tells us how the symbol is represented. For example, in the FAMILIES problem statement "married" is represented as a relation because the symbol *married* appears in atomic formulas of the form $married(term_1, term_2)$.

We say that a representation *captures* a constraint when, expressed in that representation, the constraint follows from the meaning postulates of the representation. For example, a function symbol in first order logic captures the notion of a "function"

by capturing the "single valued" constraint (among other things). More precisely, the single valuedness of a concept is captured when it is represented as a function because single valuedness then follows from the meaning postulate:[3]

$$\forall x \forall y \forall F[x = y \Rightarrow F(x) = F(y)].$$

Let us consider the sense in which a syntactic structure can be said to capture a constraint. A function symbol captures the single valued constraint with a combination of structure and behavior. The structure is function application (i.e., $F(x_1, \ldots, x_n)$) restricted to appear in the place of terms in the logic. The behavior is that of unification allowing $F(x)$ to unify with $F(y)$ only if x unifies with y.

The methodology described here takes explicit account of the fact that a combination of structure and behavior is required to capture constraints. The structures appearing in the range of the representations that my system designs have behavior associated with them. The structures capture constraints by having features that correspond to those constraints. For example, in the FAMILIES representation, a married couple is represented as a structure with two slots. The feature, two slots, corresponds to the property "married couples have exactly two individuals in them."

Structures also have procedures associated with them that provide them with behavior enforcing their constraints.[4] For example, the procedures associated with the structure for married couples interpret each slot in the structure as containing exactly one individual. It is through the combination of structure (two slots) and behavior (each slot is interpreted as containing one individual) that structures capture constraints.

The procedures of a structure enforce its constraints as instances of the structure are created and modified. For example, there is a procedure that produces a single couple from the two separate couples $< A, B >$ and $< A, C >$. This procedure enforces the fact that couples are disjoint by combining two couples when they share an individual. When the new structure is created, another procedure associated with couples makes $B = C$ to ensure that the new couple has exactly two members.

[3]This is a standard way to axiomatize first order logic with equality and function symbols. See, for example, [Bell & Machover 77]p.108.

[4]Or, put slightly differently, the procedures preserve the semantics of the representations.

Notice that if $B \neq C$ in the example above, stating that $< A, B >$ and $< A, C >$ are couples is contradictory. In this case, the structure used to represent couples must signal a contradiction to enforce its constraints. In general, when a representation captures a constraint, it signals a contradiction if a collection of facts is stated that violate the constraint.

One representation of a problem is better than another when it captures more of the constraints on the problem's concepts. For example, consider two different representations of a problem in first order logic: In the first, the concept "mother" is represented as the binary relation *mother*, while in the second it is represented as the function *mother-of*. A problem statement using the relation representation might contain the statements:

$mother(A, B)$
$\forall x \forall y \forall z [mother(x, y) \wedge mother(x, z) \Rightarrow y = z]$.

In a representation in which "mother" is represented as the function *mother-of*, the same information expressed in the two statements above is expressed in the single statement $mother\text{-}of(A) = B$ because the single valued property of "mother" is captured by *mother-of*, i.e., the general statement above reformulated in terms of *mother-of* follows from the properties of functions in first order logic.

Because better representations capture more constraints, they aid in problem solving by reducing the search space that a problem solver must consider. To illustrate, let us consider solving a very simple problem in two different representations, one capturing more constraints of a problem than the other. Suppose we are using a resolution theorem prover as our problem solver. In the first representation, the problem of interest is expressed in terms of, among other things, a relation R that is single valued. That is, the problem statement contains:

$$\forall x \forall y \forall z [R(x, y) \wedge R(x, z) \Rightarrow y = z] \quad (1)$$

Suppose further that in order to solve the problem the theorem prover must find a contradiction in the two statements:

$$\forall x \forall z \exists y [R(x, y) \wedge S(y, z)] \quad (2)$$
$$\forall x \forall z \exists y [R(x, y) \wedge \neg S(y, z)] \quad (3)$$

Finding the contradiction requires using the fact that R is single valued — that for every x there can be at most one y such that $R(x, y)$ — to determine that the negative

and positive occurrences of S resolve. The solution path for a simple resolution theorem prover will have length greater than or equal to four to reach this conclusion using (1) above. For the interested reader, Figure 1.7 gives the clausal form of this problem and one possible shortest solution.

The clausal form of the problem:
$(1')\neg R(x_3, y_3) \vee \neg R(x_3, z_3) \vee y_3 = z_3$
$(2')R(x_1, \sigma(x_1))$
$(2'')S(\sigma(x_1), z_1)$
$(3')R(x_2, \tau(x_2))$
$(3'')\neg S(\tau(x_2), z_2)$
Note that σ and τ are skolem functions.
Here is one possible shortest solution path:

Conclusion	*Justification*
(4) $\neg R(x_1, z_3) \vee \sigma(x_1) = z_3$	(2') and (1')
(5) $\sigma(x_2) = \tau(x_2)$	(4) and (3')
(6) $S(\tau(x_1), z_1)$	(5) and (2'')
(7) $\square$	(6) and (3'')

Each of the steps, except (6), is an application of the binary resolution rule. Step (6) requires a special inference rule for equality such as paramodulation.

Figure 1.7: A proof with the "single valued" property stated as an axiom.

In the second representation, the concept "R," which was represented as a relation, is represented as the function F. In this representation:

- the constraint of (1) is enforced by the theorem prover's unifier without the need for any explicit axiom
- the problem formulation becomes simply:
  $S(F(x_1), z_1)$
  $\neg S(F(x_2), z_2)$
- and the required inference is reduced to one resolution step.

1.2.2 A Model of Representation Design

In this thesis, representation design is viewed as a process of capturing constraints. The input to the process is a problem statement, the representation mapping induced

by that statement (i.e., the structure representing each concept), and the meaning postulates for a set of available structures. The system tries to select new structures to represent the problem concepts so that the constraints on those concepts are captured. As a simple example, suppose that a problem statement contains

$mother(A, B)$
$\forall x \forall y \forall z [mother(x, y) \wedge mother(x, z) \Rightarrow y = z]$

and that the available structures are those of first order logic with equality and function symbols. Then the system will change the representation of "mother" to a function by uniformly rewriting the atomic formula $mother(x, y)$ as $y = mother\text{-}of(x)$. The new statement of the problem is

$B = mother(A)$
$\forall x \forall y \forall z [y = mother\text{-}of(x) \wedge z = mother\text{-}of(x) \Rightarrow y = z]$

and the general statement follows from a suitably instantiated version of the meaning postulate for functions, i.e.,

$$\forall x \forall y [x = y \Rightarrow mother\text{-}of(x) = mother\text{-}of(y)].$$

Therefore, the problem's general statement can be removed, leaving the problem representation as simply $B = mother(A)$. For larger problems, the process considers each concept in turn, attempting to capture its constraints by redefining its representation.

The earlier discussion about the properties of good representation distinguished between capturing constraints in structure and behavior. The process described above is not concerned with whether constraints are being captured in structure or behavior, only with whether they are being captured. The representation design system's ability to design good representations is, therefore, dependent on the quality of the structures available to the constraint capturing process: they must be efficient combinations of structure and behavior. This is one reason that the research is called representation *design*. Like other designers, this process can assemble primitives (structures) to meet design specifications. However, a precondition for the process generating good designs is that the primitives be good.

Other desiderata for design primitives also apply to the available structures for representation design. For example, one desirable property of a collection of primitives is *orthogonality*. In the case of representation design, orthogonality amounts to the

collection of structures being able to capture a wide variety of constraints. This is not the case, for instance, in the example above where the only available structures are relation and function. As we will see, the representation design system described in this thesis relies on a larger, more varied collection of structures.

1.2.3 Representations and Problem Classes

One question that arises about the process described in the last section is: Should we really try to design a representation that captures every constraint of a problem, no matter how serendipitous? The advantage of doing so is that a representation capturing all of a problem's constraints can take advantage of every peculiarity of a problem to gain efficiency. However, when a representation captures every constraint of a problem it can not be used to solve any other problems.

Given a problem, my system attempts to design a representation that captures all of the constraints of that *problem's class.* Intuitively, two problems are in the same class when the same general constraints are relevant to solving them. They can differ in the individuals they mention, in the particular relationships between those individuals, or in which individuals are constrained in a particular way. For example, a problem in the same class as the FAMILIES problem would refer to the same collection of concepts (i.e., *married*, *child*, *grandchild*, *niece*, *mother*, *sister*, *brother*, *grandfather*, *sibling*). However, it could mention a different collection of individuals and could have a different number of individuals being married.

Note that since problem statements may have function symbols in them, some relationships between individuals may be specified as equalities between terms, e.g., $mother(A) = B$. Since we want to allow different problems in the same class to have different relationships between individuals, we must allow problems in the same class to have statements that equate different terms.

In defining problem class more formally, problem statements are divided into three types: those that we term *specific* because they mention only individuals (these include existentially quantified variables that are not in the scope of any universally quantified variables); those that we term *general* because they do not mention individuals (i.e., have only universally quantified variables or existentially quantified

variables in the scope of universals); and those that we term *mixed* because they contain both individuals and universally quantified variables.

To capture the intuition that specific individuals and relationships do not affect a problem's class, we abstract a problem to a class description as follows:

1. Generalize the specific and mixed statements by replacing named individuals with existentially quantified variables. For example,

   $$\forall x \neg brother(M, x)$$

   to

   $$\exists p \forall x \neg brother(p, x).$$

2. Break up conjunctive specific and mixed statements. For example, break up the statement

   $$\exists x \exists y \exists z [brother(x, y) \wedge married(y, z)]$$

   as

   $$\exists x \exists y [brother(x, y)]$$
   $$\exists x_1 \exists y_1 [married(x_1, y_1)].$$

3. Remove specific statements that are equalities.

The resulting set of statements is a class description in the sense that we could use it to generate problem instances by choosing different individuals for the existentially quantified variables. More precisely, we can create an instance as follows. For each existentially quantified statement, create one or more statements by substituting individual names for the existentially quantified variables. Then add equalities between terms in the new problem statement.

There are good reasons for designing to a problem class instead of attempting to design to a particular problem. The efficiency of the resultant representation for problem solving is very close to that of a representation that captures all the constraints of the specific problem. This representation is also reusable.

Although the definition of problem class does not explicitly refer to a problem's query, problem class is defined in terms of what is relevant to solving a problem. This depends on what is being asked. The system attempts to restrict its description

of a problem class so that it contains only statements that are relevant to solving a problem by eliminating statements that are irrelevant before it derives the class description. A precise definition of irrelevance and the system's technique for eliminating it are discussed in Chapter 3.

After eliminating irrelevance from a problem statement, the system creates two sets of statements. One set contains the specific and mixed statements of the problem. The other set contains a description of the problem class, i.e., the specific statements are generalized to replace named individuals by existentially quantified variables. The system then attempts to design a collection of structures that capture all of the constraints of the problem class.

Representation design continues until all of the constraints in a class description are captured or until the system can capture none of the remaining constraints.

To solve the problem, the specific and mixed statements are expressed in the designed representation creating instances of the structures designed from the class description. Let us call this set of structures, a *problem situation.* Because the constraints of the class are captured by the representation, those constraints are true of every problem situation that can be built with that representation. Often a constraint is captured by responding to the addition of new facts to a situation by adding still more facts. For example, the representation designed for the FAMILIES problem captures the symmetry of "married" and, therefore, when a specific statement like $married(N, P)$ is expressed in this representation, the symmetry of "married" is enforced. As a result, the situation containing $married(N, P)$ will also contain $married(P, N)$.

1.2.4 Implementation of Representations

As noted earlier, a representation is a mapping between concepts and structures with behavior. The representation design system implements structures as abstract data types (ADTs) or objects in the object programming sense. The data structure part of an ADT is used for implementing structure and the procedures are used for implementing behaviors. Access to the data structure "inside" an ADT is controlled by its procedures, preventing arbitrary manipulations of the data structure. The procedures of the ADTs that my system designs enforce constraints by maintaining

invariants in their data structure as information is added to problem situations.

For example, when "married" is represented as a relation, the system designs an ADT in terms of **sym-rel**. An instance of this ADT , denoted **married**, maintains a list of the pairs of individuals that are known to be married in a problem situation. **Married** captures the symmetry of "married" because is has a procedure that adds $< x, y >$ to the list in **married** whenever $< y, x >$ is added. The **sym-rel** ADT controls access to its internal list so that it is impossible for $< x, y >$ to get on the list without $< y, x >$ also appearing there. Thus, using this ADT, no problem situation can be constructed which violates the symmetry constraint.

In general, there are three different kinds of procedures associated with ADTs. The first kind add information to problem situations by adding to the ADT instances or creating new instances. For example, one procedure adds new pairs of individuals to **married**. The second kind of procedure enforces constraints (maintains invariants). The procedure associated with **married** that enforces symmetry is an example of this. The third kind answers queries by inspecting the structures built by ADTs. For example, another procedure associated **married** checks to see whether two individuals are married in a problem situation by searching the list for a particular pair.

1.2.5 Knowledge about Representations

The system synthesizes representations from a library of prototypical structures, or classes in the object oriented programming sense. For example, there is a library prototype called **relation**. Representations of particular relations are created by instantiating the prototype **relation**. For example, **married** is an instance defined in terms of **relation**. Here is its definition:

married: relation(family-member,family-member),

where **family-member** is the representation of the sort *family-member*.

The relations, functions, and individuals found in a problem are represented initially by instantiating the library prototypes **relation**, **function**, and **individual** respectively.

Let us now be specific about the typographic conventions that have been used. They

are summarized in Figure 1.8.

Ontological class	*Typographic convention*	*Example*
concept	quoted	"married"
symbol	italics	*married*
representation	typewriter font	`married`
prototype	typewriter font	`relation`

Figure 1.8: Typographic conventions

1.2.6 The Goal of Representation Design

The next four sections outline the four processes of representation design. The process as a whole tries to design representations that are *complete* and *maximally constrained.* Each of the four processes contributes to this goal in different ways.

A representation is *complete* with respect to a problem class just in case every problem situation in the class is representable. We assume that first order logic is complete, thus initial problem representations are always complete. The system must be sure to preserve this property as constraints are captured in a representation being designed.

A representation is *maximally constrained* with respect to a class if it captures all of the constraints of the class. This is, of course, what the representation design system tries to achieve.

The system is not always successful at designing a maximally constrained representation. Fortunately a representation that captures most of a problem's constraints is still useful because, we are still better off than we were with the initial representation. The reason is that as constraints get captured in a representation, the space of possible situations that must be searched by a problem solver using that representation is reduced. The result of representation design is a more constrained representation and a *smaller* collection of statements (the uncaptured ones) that the problem solver must reason about explicitly in that representation. [5]

[5] In effect, the problem solver uses the constrained representation to accelerate the problem solving process in the same way that specialized reasoners have been used to accelerate theorem proving. There are issues involved in the interface between a theorem prover and representations of the sort

1.2.7 Deriving an Initial Representation

Problems are stated in a sorted logic.[6] One possible description of the initial representation is the collection of data types that are isomorphic to the sort signatures of the problem concepts, e.g., since the initial statement of the FAMILIES problem contains N : *family-member*, *married* : *family-member* $\times$ *family-member*, the initial representation of the FAMILIES problem would contain

```
(deftype family-member specializes individual)

married: relation(family-member,family-member).
```

The resulting data types are a description of a complete representation because every problem situation can be expressed using them However, subsequent processes will design representations for all the concepts that appear in this description and some of these concepts may be unnecessary.

Instead of including all concepts, the system attempts to develop a description of the smallest set of concepts that constitute a complete representation. There are two types of concepts that the system attempts to exclude in this process: irrelevant concepts and redundant concepts.

Problems often contain concepts that are not relevant to their solution. We do not want to design representations for irrelevant concepts. Therefore, the system attempts to exclude irrelevant concepts from its description using a technique described in Chapter 3.

Problem statements also, at times, contain redundant information. For example, it is usually redundant to capture constraints on a defined concept if we have captured all the constraints on the concepts in its definition. For instance, if we design a representation to capture all the constraints on *child*, it is not necessary to design a separate representation for *grandchild* because it is defined in terms of *child*.

While deriving the description of the initial representation, the system tries to identify defined concepts and exclude them from the description.[7] However, a subsequent

that the system designs which I have not yet attempted to work out. However, the approach used in [Miller & Schubert 88] should work here.

[6]However, a problem's sort information may be incomplete. Chapter 3 describes a simple technique that the system has for computing sort signatures from partial information.

[7]Sometimes the choice of which concepts are primitive and which are defined is arbitrary. For

process of representation design may decide to represent a concept that is excluded at this point. The techniques that the representation design system uses to identify defined concepts are discussed in Chapter 3.

Once the representation design system has identified a set of concepts that it believes is the smallest complete representation, it uses the associated data types as its initial description. While complete, this representation does not capture any of the constraints of the problem class, i.e., situations can be created that violate any of the statements in the class description. The rest of the processes of representation design work to capture all the constraints of a problem class.

The next three sections describe the *classification*, *concept introduction*, and *operationalization* processes which attempt to design maximally constrained representations. Classification and concept introduction run as coroutines to capture as many of the constraints of a problem as possible by selecting different library structures to represent problem concepts. These processes take three inputs: the class description (derived from the problem statement after irrelevance is eliminated), the description of the initial representation, and axiom schemas defining the invariants on the library structures (ADTs). They produce two outputs: a description of the designed representations and the statements in the class description that are not captured by the representation. Operationalization then tries to capture the constraints of any remaining statements by writing new procedures and using these to further constrain the representations created by classification and concept introduction.

1.2.8 Classification

The representation design system captures constraints on a concept by representing it in terms of library structures enforcing those constraints. The structures that together capture the most constraints on a concept are identified by classifying that concept in a hierarchy of the library structures. The hierarchy is organized around the constraints that the structures enforce (see Figure 1.9). Structures enforcing

---

example, given the statement

$\forall x \forall y [child(x, y) \Leftrightarrow parent(y, x)]$

and no other general statements about either concept, one is arbitrarily considered to be defined in terms of the other.

additional constraints specialize structures enforcing fewer constraints. For example, the structure `1-1 function` specializes `function` because it enforces the additional constraint, "one-to-one"(i.e., $f(x) = f(y) \Rightarrow x = y$).

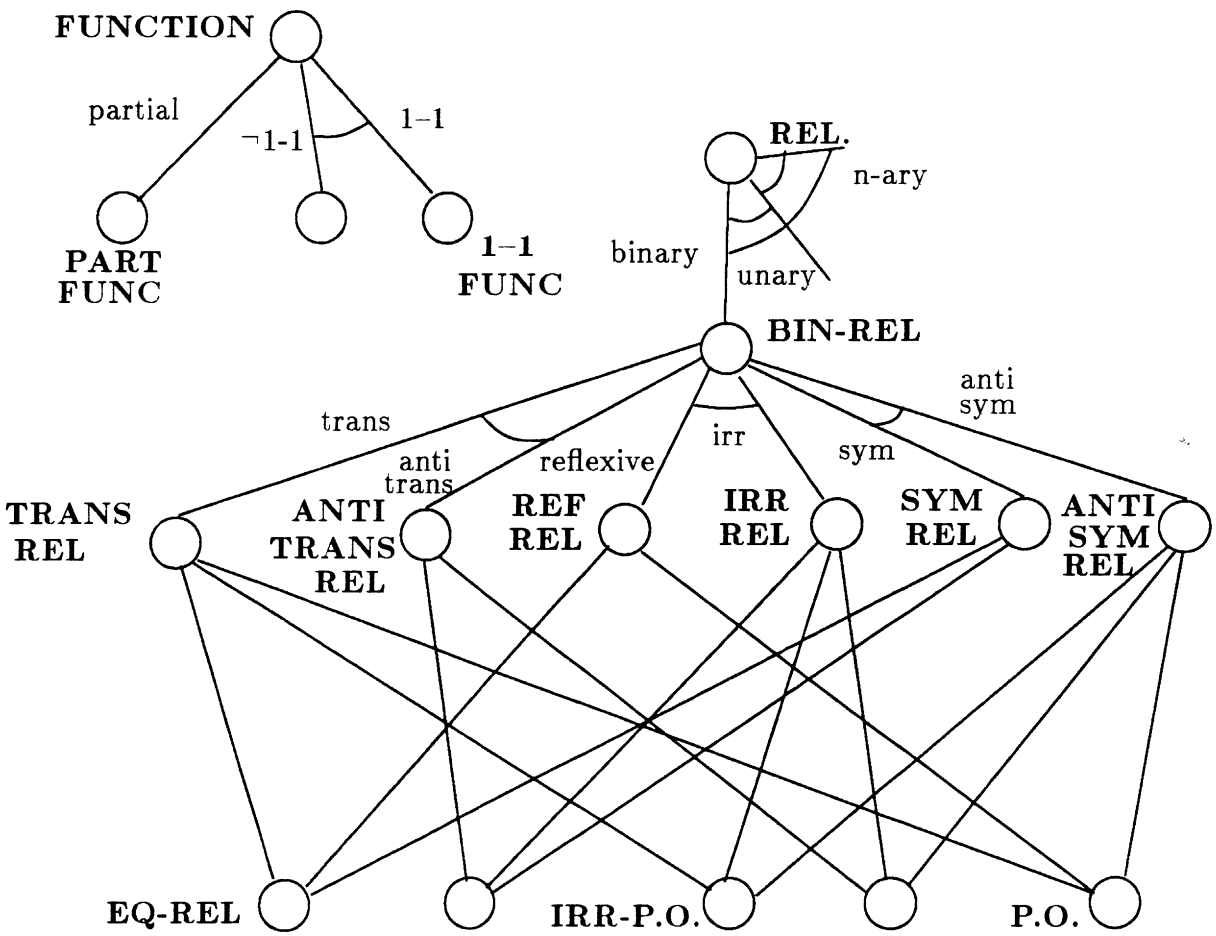


Figure 1.9: Part of the structure taxonomy

The system classifies concepts in the hierarchy, using their definitions in the initial representation to determine where to begin. For example, the representation for *married* is initially defined as

```
married: relation(family-member,family-member)
```

so the classification of *married* begins at the `relation` node.

During classification, the constraints labeling links in the hierarchy are treated as properties to show of concepts. As it reaches each node in the hierarchy, the system

looks at the properties labeling links leaving that node, trying to determine which properties the concept has. For example, the classification of *married* begins with the system looking at the links leaving the `relation` node and tries to determine if *married* is unary, binary, or n-ary. It is found to be binary from its initial representation.

The hierarchy is also annotated to show which properties leaving a node are disjoint (the arcs connecting links in Figure 4.1). For example, unary is annotated as disjoint from binary. Classification at a node is complete when the system has examined each disjoint group of properties at that node and either it has identified one property from the group or it has determined that the concept being classified has none of the properties in that group. Classification then continues with each of the nodes below the properties identified. For example, after *married* is classified as binary, classification continues with the node labeled `bin-rel`; when the system determines that *married* is irreflexive, symmetric, and antitransitive, classification completes at this node.

Classification treats unlabeled links differently. When classification completes at a node with unlabeled links leaving it, those links are marked traversable. Classification does not proceed down an unlabeled link until all other links entering the node below are marked traversable. For example, classification will only proceed from the node labeled `sym-rel` to `eq-rel` when the links leaving the nodes labeled `sym-rel`, `ref-rel`, and `trans-rel` have all been marked traversable.

To determine whether a concept has a property, the representation design system looks for a statement in the problem that indicates that the property holds. For example, to determine that *married* is symmetric, the representation design system looks for a statement of the form:

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)].$$

This technique by itself often fails because properties can be stated in many ways. The system also has a heuristic technique (discussed in Chapter 4) that tries to transform statements into a form that classification is expecting. Since the technique is heuristic, the system may fail to determine that a concept has a property even when it is stated.

When the representation design system is unable to show that a concept has any of the properties in a disjoint group leaving a hierarchy node, it asks the user about each property in turn. For example, if it is not able to determine whether or not *married* is symmetric, it asks the user (presumably the answer is "yes"). If the answer were "no," the system would ask whether it is antisymmetric.

When the system obtains positive information about a property from the user, it adds the statement of the property to the problem. For example, when the user replies that *married* is symmetric, the system adds the statement (shown above) expressing that fact.

Perhaps more interesting that the classification algorithm itself is the library of structures that it uses (part of which is shown in Figure 4.1). This library was compiled from my observations of the "structures" that people use in their diagrams when solving analytical reasoning problems. It turns out that these structures are particularly efficient combinations of structure and behavior. In addition, the constraints that they capture are applicable to a large number of analytical reasoning problems. For example, even though capturing a constraint generally requires procedures that respond to new specific facts by adding still more specific facts, clever use of structure can avoid this. In the FAMILIES problem, for instance, the structure designed for "married" captures the symmetry constraint in this way. Married couples are represented as sets of size two. This avoids having to do any extra work to enforce the symmetry constraint because, using the fact that $\{x, y\} = \{y, x\}$, the combination of the structure and procedures associated with the set representation make it the case that the statement *married*(N, P) is the same as *married*(P, N).

As constraints get captured during classification, statements expressing those constraints are identified by a process called *capture verification*, then removed from the class description. Whenever classification of a concept terminates, capture verification checks the general statements referring to the concept to see whether their constraints have been captured. The intuition behind capture verification is that a constraint is captured by a representation if it is true in every situation that can be created in that representation. Statements are checked by attempting a constructive proof using the representations of the concepts in the statement to build situations, i.e., the abstract data types designed are used to build data structures.

For example, to verify a conditional statement, capture verification constructs a problem situation representing the antecedent of the statement and then checks that situation to see if the consequent is true. If so, then the representation captures the constraint of that statement.[8] For instance, consider how capture verification checks the statement

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)].$$

A situation is created representing the antecedent, i.e., a situation in which one anonymous individual, say A, is married to another, say B. Then the situation is inspected to see if it is also the case that B is married to A. When married couples are represented as sets of size two, this proof will succeed. In this case, the system concludes that the constraint of the statement is captured.

1.2.9 Concept Introduction

Classification has a serious limitation: its success depends on the particular vocabulary used to state a problem. The FAMILIES problem, for example, is stated in terms of *married*, which is classified beginning with **relation**. None of the specializations of a **relation** capture the fact that married couples are all of size two. However, if the problem had been stated in terms of couples, classification would have been more successful because a *couple* is a specialization of **set** that takes advantage of size constraints. Concept introduction enhances classification in this example by introducing *couple* when it detects that the size constraint on *married* is not captured by **married**. This illustrates that, given a different vocabulary, classification is more successful because it allows different and sometimes more specialized knowledge in the structure library to be brought to bear in representation design.

Introduction extends the classification process by adding a new concept to a problem when classification of an existing concept does not capture all of its constraints. The introduced concepts allow the system to view an existing concept differently. For example, introducing *couple* for *married* allows the system to view "married" differently.

[8] More generally, a disjunction is verified by creating a situation in which all but one of the disjuncts is false and then checking that the last disjunct is true.

When a new concept is introduced, a corresponding representation is also introduced. For example, *couple* is actually introduced in several steps. I will use the first step to illustrate the parallel introduction of new concepts and representations. In the first step, the concept *spouses* is introduced. This is a function mapping individuals to the sets of individuals who are their spouses. A representation, `spouses` is also introduced and defined as a `function` mapping an `individual` to a `set` of `individuals`.

A new concept is always defined in terms of an existing concept in such a way that the semantics of a problem are not changed. For example, *spouses* is defined as

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

There are two reasons for introducing new concepts. The primary reason is to give the representation design system access to different representation design knowledge: the library structure representing the new concept captures different constraints and has different specializations. In the example above, for instance, since `married` does not capture all the constraints on "married," *spouses* is introduced to gain access to representations that capture different constraints.

The other reason to introduce a new concept is that it enables reformulation of the statements in the problem. This is useful because it often allows new properties to be recognized in the problem statement. Reformulation is accomplished by treating the logical definition of a new concept as a rewrite to perform on the problem statements to derive new statements. For example, when *spouses* is introduced, it is defined as

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

This is treated as a rewrite rule that derives a statement referring to *spouses* for every statement referring to *married*. For instance, this rule derives the statement $P \in spouses(N)$ from $married(N, P)$.

As new concepts are introduced, the representation design system explores a space of alternative problem formulations. For example, after *spouses* is introduced, there are two formulations: one in terms of *married* and one in terms of *spouses*. The system has a technique for estimating the cost of problem formulations so that alternatives can be compared (discussed in Chapter 5). Alternative formulations are maintained because the cost estimation technique works only when all the relevant alternative

concepts have been fully classified. For example, the formulation of the problem in terms of *married* can not be compared to the formulation in terms of *spouses* until *spouses* is fully classified.

Classification extended by introduction is called *extended classification.* Extended classification is one of the most interesting aspects of this research. While the two processes involved in it are fairly simple, the behavior of the combination can result in sequences that significantly reformulate a problem. Several examples of extended classification are given in Chapter 5; one example shows how extended classification of *married* results in the following sequence of introductions: [9]

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. `Couple` captures the following facts: not all individuals are married, each married couple is disjoint from all other married couples, married couples contain exactly two members.

The discussion so far has described how extended classification designs representations that capture general statements. The introduction process is also used to capture a special kind of mixed statement called a *restriction.* A restriction is a mixed statement that restricts the individuals that can stand in some relation to a particular individual. Here are two examples of mixed statements that are restrictions:

$\forall x \neg brother(M, x)$
(restricts the individuals that can be brothers of M),
$\forall x[child(Q, x) \Leftrightarrow x = R]$
(restricts the individuals that can be children of Q).

[9] Note that the names that the system makes up for introduced concepts are gensyms. For clarity of presentation, meaningful names have been substituted for these throughout this document. For example, I have substituted *spouses* for the meaningless name that the system gives to a concept it introduces for *married.*

By contrast, the statement

$$\forall x\ brother(M, x)$$

is not a restriction because it does not restrict the brothers of M.[10]

During extended classification of a concept, the system tries introductions that reformulate a problem so that restrictions become specific constraints. For example, during the extended classification of the **relation** *brother*, the statement, $\forall x \neg brother(M, x)$, is captured by introducing the **function** *brothers* that maps from an individual to his/her set of brothers and rewriting statements referring to *brother* so that they refer to *brothers* instead. This reformulation transforms the above statement into $brothers(M) = \emptyset$. Notice that this is a specific statement, i.e., it does not contain any universally quantified variables. Similarly, the statement

$$\forall x[child(Q, x) \Leftrightarrow x = R]$$

is captured by reformulating *child* as a **function** *children* that maps individuals to their sets of children. Then this statement becomes the specific statement $children(Q) = \{R\}$.

There are two reasons for handling restrictions in this way. First, the reformulated representation is never less efficient than the original. Second, it results in a type of generalization of the representation by introducing sets. If a problem contains a statement restricting the number of individuals that can stand in some relation to a specific individual, we would like the system to design a representation that is not dependent on the number of individuals in the given statement. This way the representation can be used for another problem that has a similar constraint, but differs in the number of individuals involved. For example, we want the representation designed for a problem containing the statement

$$\forall x[child(P, x) \Leftrightarrow x = R]$$

to also work for a similar problem containing

$$\forall x[child(P, x) \Leftrightarrow x = R \vee x = S],$$

[10]Note that there is no guaranteed syntactic technique for establishing that a statement is a restriction. Therefore, there is no guarantee that all restrictions will be reformulated as described here.

and so on.

The system accomplishes this by generalizing the sets introduced when restrictions are reformulated. For instance, to accomplish this in the above example, the system introduces the function *children*, a mapping from an individual to his/her set of children, then reformulates. The reformulated versions of the two statements above are

$$\exists y_1 \exists z_1 [children(y_1) = \{z_1\}]$$
$$\exists y_2 \exists z_2 \exists w_2 [children(y_2) = \{z_2, w_2\}]$$

which the system generalizes to

$$\exists y \exists z \; children(y) = z,$$

where z is taken to be a set. Hence, a representation is designed to allow statements restricting the set of an individual's children to a constant set of any size.

1.2.10 Operationalization

In the ideal case, extended classification maximally constrains the initial representation. Given a fixed collection of concepts, classification comes up with the most constrained collection of representations, while introduction modifies the collection of concepts when they fail to capture all of a problem's constraints. The success of extended classification depends on the collection of library structures and it can fail to produce a maximally constrained representation because there are many combinations of constraints that the library structures can not capture. When extended classification fails, the representation design system makes a final effort to maximally constrain the representation through a process called *operationalization.*

Operationalization tries to capture constraints by writing procedures that respond to any new information added to situations, in order to enforce the constraints of statements. Each procedure written for a problem statement responds to the addition of information that can violate the constraint by adding still additional information to reestablish it.

Consider an example: Suppose that in designing a representation for the FAMILIES problem, extended classification produces the concept *siblings*, a function from indi-

viduals to sets of individuals. Suppose further that the following statement has not been captured,

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

Operationalization will write a procedure that adds y to the siblings of x whenever x is added to the siblings of y. This new procedure captures the constraint of this statement whenever an element is added to a set of siblings in a problem situation.

Notice that if this operation were the only one that could change sibling sets, then the procedure described would capture the constraint of the statement above because it would ensure that no problem situation could be created in which y is a member of the siblings of x unless x is also a member of the siblings of y.

Of course, in general, there are different ways to add information to a problem situation. Therefore, operationalization must generate a procedure like the one above for every different way that information affecting a statement's constraint can be added.

It turns out that the ways information can be added to a problem situation are restricted by the kinds of operations that can add information. Operations that can add information are restricted by the procedures that already exist for the abstract data types implementing the problem representation.

For the problems that the system has been tested on, operationalization has always succeeded in capturing all constraints left uncaptured by extended classification. However, there is no guarantee of this in general. As noted, a representation that captures most of a problem's constraints is still useful in accelerating a problem solver's reasoning.

1.3 Analytical Reasoning Problems

The current system designs representations for analytical reasoning problems. These problems appear on intelligence tests like GREs and LSATs. They are intended to test a person's ability to "draw logical conclusions from information presented and to synthesize that information in order to deduce the actual structure of or interrelationships among things." [Weber83]

Analytical reasoning problems present a collection of facts about a specific situation and then ask questions that require deduction from those facts. They never ask questions requiring induction from a set of facts. There are four types of questions:

1. Is some specific fact or restriction true of a problem situation? For example, "Is it true that M has no brothers?"

2. Is it possible for some specific fact or restriction to be true in a problem situation? For example, "Is it possible for N to be the brother of M in the current situation?"

3. Find all the individuals that stand in some relation to a specific individual in a problem situation. The FAMILIES problem question is an example of this type: "Find all the individuals that are siblings of S." This type of question may also ask if the individuals found are all the individuals that can have this property or are just all that can be found in the problem situation. For example, a question may ask us to find all the siblings of an individual and then ask whether it is possible for there to be others.

4. Find *the* individual that stands in some relation to a specific individual. This question assumes there can be only one such individual. For example, "Find the mother of S."

The representation design system accepts problems with the types of queries described above and designs representations specifically to answer those queries. A problem query can influence representation design in several ways. For example, the system usually reformulates problems so that find-all queries are translated into find-the queries because find-the queries can be answered more efficiently. For instance, the FAMILIES problem is initially stated in terms of the relation *sibling*. Given the query, "find all the siblings of S," the system reformulates the problem in terms of *siblings*, a function mapping an individual to his/her set of siblings so that the query becomes, "find the sibling set of S."

The system also reformulates problems that have queries asking about mixed facts because these, too, are often very expensive to answer. For example, if instead of the statement $\forall x \neg brother(M, x)$, the FAMILIES problem contained the query, "Is it

necessarily the case that M has no brothers?" the system would still reformulate the problem in terms of brother sets.

Even though analytical reasoning problems do not require it, the system can also answer necessity and possibility queries about general facts. We now explain how each of the different types of queries is answered for statements of the form

$$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)).$$[11]

Queries are converted to one or more statements in this form when they are not stated as such.

1. To determine if some unconditional (i.e., $n = 0$) specific fact is true in a problem situation, the system inspects the situation to see if the fact is present. If it is, then the system answers yes; otherwise it responds that it does not know.

   To determine if a conditional specific statement is true, the system extends the situation adding facts that correspond to the antecedents (i.e., each of the ϕ_i) and then, if the consequent (i.e., the conjunction of the ψ_i) is true, it answers yes. For example, suppose the system is asked to show that the following fact is true:

   $$married(A, B) \Rightarrow sibling(A, C),$$

   where A,B, and C are individuals in a situation. It does this by adding $married(A, B)$ to the situation. If $sibling(A, C)$ is true in the new situation, the system answers yes. Otherwise, it says that it does not know.[12]

   To determine if a statement involving universal quantification is true, the system follows the procedure used by capture verification. To check an unconditional statement, the system extends the situation by creating anonymous individuals for each of the universally quantified variables in the statement. If the unconditional statement is true of those individuals in the new situation, the system answers yes.

[11] Note that we could just as well have used disjunctive normal form, i.e., we could replace this statement with m statements of the form

$$\neg\phi_1 \vee \cdots \vee \neg\phi_n \vee \psi_i.$$

However, it is more natural to explain representation design for statements in this form.

[12] As explained in Chapter 4, there is one other case. The system also answers the question yes if the situation containing $married(A, B)$ is contradictory.

To check a conditional statement, the system proceeds as though checking a specific conditional statement except that it creates anonymous individuals for the universally quantified variables.[13] For example, suppose the system is asked to show that the following statement is true in a situation:

$$\forall x[child(S, x) \Leftrightarrow (x = A \vee x = B \vee x = C)].$$

This is the question, "Are A, B, and C the only children of S?" To establish this, the system converts it to two statements (in the form described above) corresponding to the two directions of the biconditional. Then it does a constructive proof corresponding to each of these statements.

2. To determine if it is *possible* for some unconditional specific fact to be true in a problem situation, the system adds that fact to the situation. If a contradiction results, the system concludes that the fact is not possible, otherwise it responds that it does not know. A similar procedure is included for conditional specific facts. As with necessity questions, possibility queries with universal quantification in them are handled by replacing the variables with anonymous individuals.

3. To find all the individuals that stand in some relation to a specific individual, the system searches the problem situation looking for all facts relating individuals to the specific individual. To determine if the individuals found are the only individuals that can stand in that relation to the specific individual, the system constructs and attempts to prove an appropriate mixed query. For example, suppose the query is "find all the children of S" and the system finds that A, B, and C are all children of S. Then, to answer the second part of the query it tries to prove that A, B, and C are the only children of S, as described in 1 above.

4. A find-the query can only ask for the value of a functional expression like "find-the $siblings(S)$." These questions are answered by retrieving from the problem situation the image of an individual under the `function`. For example,

[13] If there are universally quantified variables appearing in the consequent that do not appear in the antecedent, then corresponding anonymous individuals are created and the system checks that the consequent is true for these.

the query, "find-the *siblings*(S)," is answered by retrieving from the problem situation the image of S under **siblings** (the representation of *siblings*).

As noted above, the representation design system can transform find-all queries into find-the queries by reformulating a problem. The reformulation has the effect of adding a new kind of individual to the problem. For example, the query "find-all $x \mid sibling(S, x)$" is transformed into "find-the *siblings*(S)," where the range elements of *siblings* are sets of individuals. One advantage of reformulating in terms of a find-the query is that **sets** can be marked to indicate that all members are known. This makes answering the second part of a find-all query much easier.

1.4 Soundness of Representation Design

One question that arises about the representation design process concerns how much faith we should place in the answers that we get with representations designed by the system. This is the question of whether or not the representation design process is sound.

I have shown that if the representation design system produces a fully constrained representation and that representation halts while building a problem situation, then the answers produced with that representation are always answers to the original problem.

There are two kinds of lemmas that must be proved to obtain the soundness result. First, since representation design changes problem statements, we must prove that all transformations done on the predicate calculus problem statement are sound. Of particular interest is the process of introduction because it adds new concepts to a problem. Chapter 5 gives a soundness condition for introductions. I have shown that all introductions in the system meet this condition and, therefore, the introduction process as a whole is sound.

The purpose of the second kind of lemma is to show that the process of capturing constraints in a representation is sound. In particular, we must show that when the constraint of a general statement is captured in a representation, the representation

can be used only to create situations in which the constraint is true.

1.5 Scope and Limitations

The representation design system is knowledge based and, as with other such systems, it is very difficult to formally characterize its scope of applicability. This section, therefore, attempts to converge on this issue from two perspectives. First, we give evidence indicating that the current implementation provides the framework and much of the knowledge required to handle most analytical reasoning problems. Second, we roughly characterize the scope of applicability of the current system and give examples of problems to which the system is not applicable.

1.5.1 Coverage

Section 1.3 provided an informal characterization of the class of analytical reasoning problems. I believe that the current system can easily be extended to provide reasonably high coverage of these problems. I characterize the coverage of the existing knowledge by answering the following two questions: First, how well does the existing library cover a large body of analytical reasoning problems? Second, how much overlap is there in the knowledge used on different problems?

There is evidence to suggest that the current structure library population is very close to a sufficient collection for designing representations for most analytical reasoning problems. The current system was designed after a survey of approximately two hundred analytical reasoning problems. From these, I compiled a set of twenty representative problems. From the representative set, I chose three problems that I found among the most difficult to solve. I call these the *paradigm* problems. I then studied the representations that I and two other people used in solving the paradigm problems. The current structure library population is the result of that study.

Examination of the other seventeen representative problems showed that the existing structure library was sufficient: the problems did not use any additional structures.

This research would not be interesting if each problem solved used a disjoint subset of

the knowledge. This has proven to be far from the case. The current representation design system generates representations for the three paradigm problems (shown in Figure 1.2, Figure 1.5, and Figure 1.6) and three variations (twelve problems total).

Even though the paradigm problems appear quite different, the representations generated for them use only structures from the existing library. Furthermore, Figure 1.10 shows that most of the structures are used in more than one problem. Thus, there is significant overlap in the structure library knowledge used in designing representations for the paradigm problems.

Problem	*Structures Used*
FAMILIES	`function`, `1-1 function`, `partial-function`, `partial-1-1-function`, `disjoint-set`, `fixed-size-disjoint-set`
WAITERS	`function`, `fixed-size-set`, `set` `antitrans-antisymm-irref-rel`
PROFESSORS	`1-1-function`, `1-1-partial-function`, `set`, `fixed-size-set`

Figure 1.10: Library structures used in each paradigm problem.

There is also overlap in the introduction knowledge used in generating representations for these problems. There are a total of eleven rules in the introduction knowledge base. Figure 1.11 shows the number of different rules used in designing each paradigm problem and the number of total rule firings that occurred in the design efforts. All of the rules in WAITERS were also used in the other two problems. Four of the rules in PROFESSORS were used in FAMILIES.

Problem	*# Different Rules*	*# Total Rule Firings*
FAMILIES	9	31
WAITERS	3	7
PROFESSORS	5	11

Figure 1.11: Data on introduction rule usage in paradigm problems.

1.5.2 Scope of Applicability

The current version of the system designs representations for problem classes. A class description is a set of first order statements and a set of queries about specific or mixed facts. Clearly, the system's applicability is limited to classes of problems that can be stated in this language. An example of a problem that can not be stated naturally in this language is one requiring a proof by induction.

There are also problem classes that can be stated but for which the system designs an *incomplete* representation. A representation designed for a problem is incomplete when it captures all of the constraints on the problem but it can not be used to reliably answer the types of queries given in the problem statement. For example, a representation for a problem is incomplete if that representation captures all of the constraints on the problem, the problem contains the query $\Box\phi$, ϕ follows from the problem, but the system answers the query with unknown.

Note that the notion of a complete representation is much weaker than the notion of a complete deductive system for first order logic because queries are restricted to ask about specific or mixed facts with all mentioned individuals being named individuals. This means that a representation can be complete without being able to correctly answer queries about general statements or statements with existential quantification.[14]

One source of incompleteness is that, in general, the system can not design representations for disjunctive problem situations. The difficulty is best illustrated by an example. Consider the following propositional problem:

$W \vee T$
$R \vee S$
$(W \wedge R) \Rightarrow Q$
$(W \wedge S) \Rightarrow Q$
$(T \wedge R) \Rightarrow Q$
$(T \wedge S) \Rightarrow Q$
Query: $\Box Q$.

Q follows in this problem. However, the representation that the system designs will

[14]Note also that even though section 1.3 discusses using representations to answer queries about general facts, these are not included in the notion of a complete representation and we will never guarantee their completeness.

not determine this. The difficulty is that there is no way to create a problem situation representing $W \vee T$ without committing to one of them actually being true. In this problem, the truth value of Q depends only on $W \vee T$ and $R \vee S$ being true.

One way to handle disjunction like this is to use the representation that the system designs to reason by cases. For example, a problem solver could use the above representation by assuming $W \wedge R$ and noting that Q follows; then assuming $W \wedge S$ and noting the same; and so on. By trying all possibilities, it could establish that Q follows. One reason that the system does not design representations that reason by cases to enforce constraints is that doing so requires exponential time.

Note that, even though the system does not design representations for arbitrary disjunctive situations, it has special purpose techniques for designing representations for certain restricted forms of disjunction.

1.6 Related Work

This section highlights the differences between the research reported in this thesis and previous work in several areas. A more thorough treatment can be found in Chapter 8.

1.6.1 Solving Word Problems

One might expect there to be an interesting relationship between my system and previous systems that solve word problems. However, most work in this area has been concerned with translating an English problem statement to a pre-established target representation. For example, the object of STUDENT [Bobrow68] was to translate a high school level algebra word problem into a system of algebraic equations. Also, Bobrow was primarily concerned with natural language problems and I have not addressed this.

1.6.2 Automatic Programming

Most automatic programming systems perform tasks such as algorithm design, data structure selection, and optimization (of algorithms and data structures) within a fixed representation which is chosen by a person prior to the point where the system gets involved. By contrast, my system is concerned with earlier steps in the problem solving process during which a representation is designed. To illustrate this difference, consider the work on data structure selection reported in [Barstow79]. This system would, for example, select a data structure for implementing some set described in a program specification. By contract, my system is concerned with identifying the "right" sets with which to represent the problem. In most problems, it introduces new sets because they can be implemented more efficiently. Furthermore, my system chooses the same data structure to implement all sets. Thus it has an entirely different level of concern.

Many efforts in automatic programming have generated a program from a formal specification including axioms describing the desired program and the programming language operations available to implement that program. As many researchers working in this area have pointed out, the formulation of these axioms can have a dramatic effect on whether this approach succeeds. Searching for better formulations of a set of axioms is a large part of what my research is about.

1.6.3 Good Representation

As I have already stated, the principle difference between my research and previous efforts to understand what makes a representation good is that they were concerned with recognizing the properties of good representations, while my research is about generating such representations prospectively.

1.6.4 Problem Reformulation

In some ways my work can be seen as an extension of [Korf80]. Korf was concerned with characterizing a space of possible representations and types of transformations on representations. My work is concerned with *how to choose the right transformations*

to do to arrive at a good problem representation. I have identified some of the essential properties of representations and given a method to design representations with those properties.

Korf (and Amarel[Amarel68]) viewed problem solving as state space search and observed that changes in representation (i.e., the description of a problem state) affect the size of the space. The focus of my work has been explaining *how* representations do this and how to design representations that yield smaller search spaces. The claim is that one way to reduce a problem's search space is to design a representation that captures more constraints in its structure and behavior.

1.6.5 Specialized Reasoners

There is a body of work whose concern is to develop a framework in which multiple specialized reasoners can be used together in problem solving (see, for example, [Brachman et.al84, Miller & Schubert 88]). Each reasoner is specialized to perform certain inferences efficiently and together a collection of such mechanisms can be used to accelerate a general problem solver (usually a theorem prover). My research complements this work because my system can be viewed as designing specialized reasoning systems.

1.7 Reader's Guide

The next chapter explains the three evolving descriptions that the representation design system maintains throughout the design process. One can think of these as descriptions of the problem statement, the representation being designed for that problem, and the relationship between the first two descriptions. The chapter also describes a language for defining and instantiating abstract data types. This is used by a person defining library structures and by the system in introducing new representations.

Chapter 3 describes how the system derives a description of a problem's initial representation. This is done in three steps:

1. The primitive concepts are identified from a problem statement.
2. Irrelevance is removed from the problem.
3. The representation of each relevant primitive is added to the description.

Chapter 4 discusses the classification process and the knowledge used in the process. Chapter 5 discusses the concept introduction process, explains how classification and introduction are integrated as extended classification, and gives detailed examples from extended classification in the FAMILIES problem. Chapter 6 discusses the operationalization process.

Chapter 7 details the way representations are implemented. First, it explains the equality system relied on by representations and a mechanism for creating anonymous individuals. It explains the behavior of the library structures `individual`, `relation`, `function`, and `set` and how they integrate with the equality system. An example is given of the system creating a specialized representation.

Finally, Chapter 8 discusses related work and Chapter 9 provides a summary and a discussion of future work.

Chapter 2

Descriptions Used in Representation Design

The next four chapters discuss the representation design process in more detail. In preparation for this, this chapter explains the descriptions that the representation design system maintains throughout the design process. One can think of these as three evolving descriptions: one of the problem statement, one of the representation being designed for that problem, and the third a description of the relationship between the first two.

The *problem statement language* (PSL) is a sorted first order logic. The *representation description language* (RDL) is a collection of constructs for defining abstract data types implementing representations and representation prototypes.

2.1 The Problem Statement Language

Problems consist of a collection of statements and a collection of queries. The statements include sort declarations for the domain individuals (e.g., $N : family\text{-}member$) and statements in the logic. The logic is sorted and contains the distinguished relations $\in$ and $=$. For convenience, appropriate syntax is included so that terms can denote extensional and intentional sets. Extensional sets are denoted by terms of the form, $\{t_1, \ldots, t_n\}$, where each of the t_i are terms. Intentional sets are denoted by

terms of the form, $\{x \mid \phi\}$, where ϕ is a formula in which x occurs free.[1]

The query language allows necessity, possibility, find-all, and find-the queries. Necessity queries, written $\Box\phi$ (where ϕ is a specific or mixed statement in the sorted logic), ask if a given fact is true in a problem situation. Possibility queries, written $\Diamond\phi?$, ask if a fact is consistent with a problem situation. Find-all queries, written find-all $x \mid \phi$ (where x is free in ϕ), ask for all individuals in a problem that stand in some relation to a specific individual. The FAMILIES problem query is an example of this type:

find-all $x \mid sibling(S, x)$.

In addition, an answer to a find-all query states whether the individuals found are *all* the individuals for which ϕ is true, or just all the individuals that could be found given the problem statement. Find-the queries are written find-the τ, where τ is a function application term in the logic. These queries ask for the individual that the function application denotes in a problem situation.

2.2 A Language for Defining Representations

Representations and prototypes are implemented as abstract data types (ADTs). The representation design system has a language for defining ADTs. It is used in two ways: by a person to define structure prototypes and by the representation design system to design new representations in terms of prototypes. Prototypes are implemented as *parameterized ADTs*, for example, the library structure `relation` is parameterized so that instances can be created with different arities and over different sorts of individuals.

Library prototypes are defined by a person using the **deftype** construct. Its form is:[2]

[1] These extensions to the syntax are not formally required. We can, for example, replace the term $\{x \mid \phi\}$, by a variable s and then include the formula $\forall x[x \in s \Leftrightarrow \phi]$.

[2] The syntax used here is a stylized version of the actual syntax used in the implementation. The actual syntax is similar to LISP flavor syntax.

```
(deftype type-name (parameter-list)
         [declaration-list]
         [proc-list])
```

Type-name is a symbol that names the type being defined. **Parameter-list** is a list of formal parameters that are bound to particular types and constants when instances or subtypes of **type-name** are defined. The **declaration-list** defines the structure part of the new type. It is a list of instance definitions. Thus new types are defined with existing types as components. **Proc-list** is a list of definitions for procedures that operate on the structure components to provide the abstract operations of the new type. For example, here is part of the definition of the library type `relation` (some of the procedures have been omitted):

```
(deftype relation (dom1:type &rest domains:  list(types))
         [relation-list:  list(tuple(dom1 &rest domains))]
         [(defproc add-relationship (rel:  tuple(dom1 &rest domains))
                   (push rel relation-list))
         (defproc related?(rel:  tuple(dom1 &rest domains))
                  (member rel relation-list))])
```

The type `relation` takes as parameters a list of n domains and produces a representation of an n-ary relation. For example, this type is instantiated to define `married`. The domains in the parameter list are the names of types representing sorts. The data structure part of `relation` is a list of n-tuples. `List` is another type defined in this language. It has a `push` operation and a `member` predicate. Representations defined as `relations` store n-tuples of individuals that stand in a relation in a particular problem situation. For example, when creating a problem situation for the FAMILIES problem, new pairs like $< N, P >$ get added to the relation list inside `married`. This is done by executing the procedure add-relationship with $< N, P >$ as its argument.

The RDL has facilities to define prototypes that inherit structure and behavior from other prototypes. For example, one can define a prototype `sym-ref-rel` for symmetric reflexive relations in terms of a prototype for symmetric relations and a prototype for reflexive relations.

The system designs representations for concepts and sorts in a problem. Representations of concepts are designed by instantiating library prototypes. The RDL has a construct for this purpose which has the form:

instance-name: type-name(parameters).

As an example, `married` is defined as

```
married: relation(family-member,family-member).
```

Representations of sorts are designed as subtypes using one of the following constructs:

```
(deftype new-type specializes type-name(parameters))
(deftype new-type specializes type-name(parameters)
disjoint).
```

For example, there is a library prototype called `individual`. A representation is designed for the sort *family-member* as a subtype of `individual`:

```
(deftype family-member specializes individual).
```

Instances of this type are used to represent individual family members like Q. Subtype definitions have the standard semantics: every instance of a type is also an instance of its supertypes.

The second form of the subtype construct is used to define disjoint subtypes, i.e., subtypes of a type T_1 that are disjoint from all other defined subtypes of T_1. For example, we can define the following disjoint subtypes of `family-member`:

```
(deftype male-fm specializes family-member disjoint)
(deftype female-fm specializes family-member disjoint).
```

Once a type is defined to be a disjoint subtype of some type T_1, all other types defined as subtypes of T_1 are assumed to be disjoint. Hence, once `male-fm` is defined as above, all other types defined as subtypes of `family-member` are assumed to be pair disjoint from each other and from `male-fm`.

The system always defines types to represent domain individuals as disjoint subtypes of `individual`. We did not do this in the example above because there is only one type of domain individual in the FAMILIES problem and thus in this case it does not matter which form of definition we use.

2.3 Maintaining the Representation Mapping

The system maintains the representation mapping for concepts by maintaining a data type definition for the representation of each concept, e.g.,

`married: relation(family-member, family-member).`

One part of the design process involves selecting more specialized library types for representing concepts and redefining their representations in terms of those more specialized types. The other part of representation design involves introducing new concepts (and reformulating the problem). As new concepts are introduced, representations get defined for them.

An integral part of this process is the introduction of new sorts. We describe the role that sorts play in the representation design process by first reviewing the general role of sorts in logic and automatic theorem proving and then showing that the representation design system uses sorts in much the same way as a person developing a sorted formulation of a problem.

It is well known that a sorted logic is equivalent to an unsorted logic containing a unary relation symbol for each sort, but that theorem proving in a sorted logic can be significantly more efficient [Cohn88]. There are two reasons for this. First, specialized inference rules are known that enforce relationships between sorts by restricting the sorts of objects that can unify. One can view these specialized inference rules as capturing taxonomic constraints between sorts. Second, sorted formulations are typically simpler than unsorted formulations. This is because some constraints between concepts are re-expressed as relationships between sorts and handled by the sort machinery. Therefore, the logical statements expressing those constraints are removed from the problem. Also many of the remaining statements simplify because sort predicates are removed. For example, in an unsorted formulation of the FAMILIES problem, the symmetry of *married* might be stated

$$\forall x \forall y [family\text{-}member(x) \wedge family\text{-}member(y) \Rightarrow (married(x,y) \Rightarrow married(y,x))],$$

while in the sorted version this statement is simplified to

$$\forall x \forall y [married(x,y) \Rightarrow married(y,x)],$$

given $married : family\text{-}member \times family\text{-}member$.

When formulating a problem in a sorted logic, a person declares sorts, specifies relationships between those sorts, gives the sort signature of each problem concept, and then formulates sorted versions of the problem statements. Sorted logics differ in the relationships allowed between sorts; typically taxonomic relationships are specified. We will call the collection of relationships specified between the sorts in a problem the problem's *sort structure*. For example, in a problem concerning several different kinds of animals, one might specify the sort structure as the sort/subsort taxonomy shown in Figure 2.1.

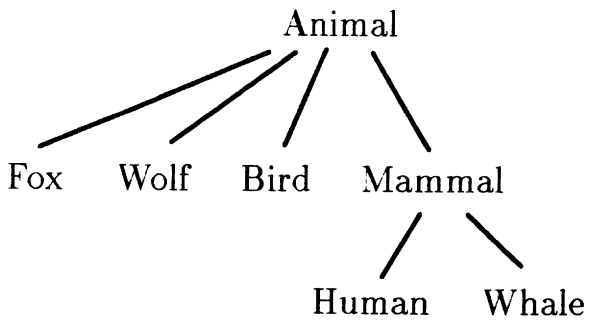


Figure 2.1: A sort tree for a problem about animals.

There is always more than one way to sort a problem. Different sort structures capture different relationships between sorts and lead to different formulations of a problem. For example, the formulation of a problem in terms of the sort structure in Figure 2.1 captures a number of constraints in the sort structure, e.g.,

$\forall x[Fox(x) \Rightarrow Animal(x)]$,

while a sort structure that did not include one of these sorts would not capture the above constraint.

Clearly, a skilled person formulating a problem in a sorted logic will attempt to find the formulation that captures the most constraints in the sort structure.

Like the skilled person, the representation design system incrementally builds up a more sorted formulation of an input problem as design proceeds. The introduction of new sorts is the system's mechanism for picking out sets in a problem to design representations for: As new sorts are introduced, representations are designed for them. This allows the system to capture two kinds of constraints: sort/subsort

constraints and constraints between or common to the elements of a sort. It captures sort/subsort constraints by building a type taxonomy with the representations of sorts. There are special library structures for capturing constraints between elements of a sort. The system captures such constraints by designing representations for sorts in terms of these special structures.

Normally, when formulating a problem, a set of sort symbols is introduced and then concepts (e.g., relations or functions) are defined in terms of these. The system introduces concepts and sorts simultaneously. Every new concept is logically defined in terms of an existing concept. Section 1.2.9 explained how such definitions are used as rewrite rules. Logical definitions of new concepts are also used to derive new sorts.

When an introduced concept is a relation, the system treats it as a predicate defining a new sort: The sort defined is the subsort of the relation's domain for which the relation is true. For example, the following statement introduces the relation *couple*:

$$\forall x \forall y [couple(\{x, y\}) \Leftrightarrow married(x, y)].^3$$

It is treated as defining a new sort, which I will call *married-couple*, whose individuals are those doubleton sets (of family members) for which *couple* is true.

When an introduced concept is a function, the system defines a sort for its range elements. For instance, when designing a representation for the FAMILIES problem the system introduces brother sets by introducing *brothers*, a function mapping an individual to his/her set of brothers. It is given the following logical definition:

$$\forall x \forall y [x \in brothers(y) \Leftrightarrow brother(y, x)].$$

This is taken as defining a new sort for the range elements of *brothers*, call this sort *brother-set*. *Brother-set* is a subsort of *set(family-member)*, i.e., brother sets are sets of family members. A representation is defined for *brother-set* as

`(deftype brother-set specializes set(family-member))`,

making `brother-set` a subtype of `set(family-member)`. Also a representation is defined for *brothers* as

`brothers: function(family-member,brother-set).`

[3]For clarity of presentation, we will always write logical definitions with the new concept appearing on the left hand side.

This is an unusual way to define a sort. Normally we would define not define a sort for brother sets, instead we would give the signature of *brothers* as $family\text{-}member \rightarrow set(family\text{-}member)$. However, one of the reasons for introducing *brothers* is to investigate the structure of its range in an attempt to design a specialized representation for it. The system's mechanism for doing this is to define *brother-set* to be exactly the range of *brothers*. In exposition, we will often call such a sort a range and call the associated function its defining function.

Note that while defining sorts like *brother-set* for the ranges of functions is unusual, it is equivalent to defining a sort predicate and then defining the function with the new sort. For example, we can define the sort predicate *brother-set* as

$$\forall x[brother\text{-}set(x) \Leftrightarrow \exists y(y \in \{z \mid brother(x, z)\})]$$

and then specify the signature of *brothers* as $family\text{-}member \rightarrow brother\text{-}set$.

Chapter 3

Deriving an Initial Representation

This chapter explains how a description of the initial representation is derived from a problem statement. The processes described here result in a complete representation that is the starting point from which a maximally constrained representation (preserving completeness) is designed.

The goal in deriving a description of the initial representation is to identify a set of concepts from which to design the specialized representation. This set, which we will call the set of *represented* concepts, should be sufficient to express all the constraints of a problem class. In addition, it is desirable that it include only concepts that are necessary for expressing the constraints of a problem class and responding to the query.

We first discuss a sufficient condition. A *primitive* concept is one that is not defined in terms of other concepts.[1] To be sufficient, the set of represented concepts should include all the primitive concepts that are relevant to solving the problem. Primitive concepts are required because a representation can not be designed to capture all the relevant constraints without them. Suppose a relevant primitive concept is left out. Since this concept is relevant, it must have constraints defined on it that are used in solving the problem. But since the concept is not represented and not definable in terms of represented concepts, a representation can not be designed to capture those constraints.

[1]The set of primitive concepts of a problem may not be unique. When concepts are mutually defined, one is chosen arbitrarily as a primitive.

Relevant concepts with restrictions[2] on them are included in the representation description even if they are not primitive. For example, the statement,

$\forall x \neg brother(M, x)$,

causes *brother* to appear in the set of represented concepts for the FAMILIES problem even though it is not a primitive.

These concepts are included in recognition of a fact about the representation design process: The strategy that the representation design system uses to capture restrictions is to introduce alternative concepts that transform the restrictions into specific constraints. For example, to capture the constraint of the statement above, the concept *brothers* is introduced and the statement is transformed into the specific statement,

$brothers(M) = \emptyset$,

where *brothers* is a function from an individual to his/her set of brothers.

Such introductions are done by extended classification. Since the only concepts that get classified are those that are represented, concepts with restrictions on them are included so that they will get classified.

Concepts appearing in find-all queries are also included in the representation description even if they are not primitive. Like defined concepts with restrictions on them, concepts appearing in find-all queries are included so that they will be subjected to extended classification. Extended classification of these concepts will usually cause the problem to be reformulated so that the find-all query becomes a find-the query. For example, *sibling* is included in the representation description of the FAMILIES problem because it appears in the problem's find-all query. Extended classification of *sibling* introduces the function *siblings*, mapping an individual to his/her set of siblings. This allows the problem query to be reformulated as

find-the $siblings(S)$.

Save the exceptions just discussed, it is desirable that the set of represented concepts include only primitives. The basic reason for this is economy of mechanism. Since

[2]Recall that a restriction is a mixed statement that restricts the individuals that stand in some relation to a specific individual.

instances will be designed for all the represented concepts, representing more concepts than are strictly necessary can cause unnecessary machinery to be designed. This will not only cause the representation design system to do more work than necessary but it will also cause extra work to be done in maintaining redundant constraints when the representation is used.

Two types of concepts are considered unnecessary: those that are irrelevant to answering a problem's queries and those that are not primitive, do not have relevant restrictions on them, and do not appear in find-all queries.

When a defined concept is represented, redundant general constraints will be maintained by the resultant representation. For example, suppose *brother* is defined as

$$\forall x \forall y [brother(x,y) \Rightarrow sibling(x,y) \wedge male(y)]$$

and that the concepts *sibling* and *male* are represented. Designing a separate representation for *brother* will cause its constraints to be captured redundantly because the representation **brother** will be designed to capture the constraints on *brother*. For instance, **brother** will capture the irreflexivity of *brother*. But notice that **sibling** will capture the irreflexivity of *sibling* and since $brother(x,y) \Rightarrow sibling(x,y)$, the irreflexivity of *brother* will be captured twice.

We now proceed to discuss the four steps that identify the set of represented concepts:

1. An initial set of primitive problem concepts is identified. As we will see, when there is missing definitional information, it is acquired in this step.

2. Concepts that are irrelevant are removed from the problem statement and from the set of represented concepts. The procedure that does this is sound but not complete: Only irrelevant concepts will be eliminated but it is possible for it to miss irrelevant concepts. Irrelevance is identified after the primitive concepts because the additional definitional knowledge acquired in step one can change what is relevant.

3. Concepts that have explicit restrictions on them are identified and added to the set. Concepts that have implicit restrictions on them (i.e., those for which a restriction follows from the problem but is not stated initially) are identified later in representation design.

4. Concepts appearing in find-all queries are added to the set of represented concepts.

A description of the initial representation is constructed by defining representations for the concepts identified in the four steps above. The initial representation of each concept is isomorphic to its sort signature.

The next three sections of this chapter describe how steps 1–3 above are performed. Step four is straightforward. Derivation of the initial representation from a problem's sort signature is also straightforward. However, problem statements often give only partial sort information. The last section discusses some simple techniques that the system uses to complete a sort signature when only partial information is given.

3.1 Identifying The Primitive Concepts of a Problem

A concept is considered to be defined when there are equivalent necessary and sufficient conditions for it. The simplest case of this occurs when a concept appears alone on one side of a biconditional. For example, the following is a definition for *brother*:

$$\forall x \forall y [brother(x, y) \Leftrightarrow sibling(x, y) \wedge male(y)].$$

Of course, the simple syntactic strategy of looking for such biconditionals will miss concepts with necessary and sufficient conditions that are semantically equivalent but syntactically different. For example, if we take a definition and break it up into two implications, the simple strategy will no longer recognize it.

The system uses is a more general technique that depends on a connection graph built during the irrelevance analysis. This technique is described below in section 3.2.3. Here we point out that even the more general technique will sometimes fail to identify definitions. However, this never causes the system to design an incorrect representation; it just causes the system to design extra machinery enforcing constraints redundantly.

Identifying primitive concepts can be complicated by the fact that real problems are *incomplete*. An incomplete problem is one that does not supply sufficient information

to answer its queries. When a problem is incomplete, we can not determine whether a concept is primitive since its definition may simply be missing. In recognition of this possibility, the representation design system asks the user whether he/she would like to define any of the concepts that are primitive in the problem statement. Acquiring definitions for such concepts often uncovers additional concepts that did not appear in the problem statement. The process of prompting for new definitions continues until the user declines to further define any of the concepts that the system believes to be primitive.

Acquiring definitional information is one of the techniques that the representation design system uses to try to complete problem statements. It is clear that this technique is at the mercy of the user. There is nothing the system can do if he/she declines to provide a definition that is required to complete the problem statement.

3.1.1 Section Summary

Throughout the rest of this chapter and the next three, we will summarize each section by showing how the processes described in it change the statement of the simplified FAMILIES problem shown in Figure 3.1.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all $x \mid parent(S, x)$

Figure 3.1: A small problem about families.

The process of acquiring missing definitional knowledge expands this problem statement to the one shown in Figure 3.2.

In addition, *child* and *married* have been identified as primitives. Note that *parent* could have been identified as primitive instead of *child*. However, both of these can not be primitive because of the biconditional between *child* and *parent*. Note also that the choice of one of these as primitive does not affect the representation that the system designs.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
Query: find-all $x \mid parent(S, x)$

Figure 3.2: The example problem with definitions added.

3.2 Eliminating Irrelevant Information

It is desirable to eliminate irrelevant information before a representation is designed because we do not want representations to capture irrelevant constraints. Part of my approach is to design representations for problems before solving them.[3] In light of this, I have developed an irrelevance filter that does not require a problem to be solved to use. However, it does not always eliminate all irrelevance.

The filter is guaranteed not to eliminate relevant information as long as it is run on a complete problem statement. However, the system runs the filter right after primitive concepts are identified. Subsequent steps in representation design may further complete a problem statement and, therefore, running the filter at this point may cause relevant information to be eliminated. Therefore, the system allows for the possibility that information filtered out of a problem statement may later be "resurrected."

The discussion that follows begins by defining the class of irrelevance that is usefully eliminated in representation design. The filter is described and proved sound. Also it is shown to be incomplete.

3.2.1 Strong and Weak Irrelevance

We define two classes of irrelevance: strong and weak. A fact is *strongly irrelevant* to a problem P if it can not be used to derive the answer for any problem in P's class.

[3] An alternative approach is to solve a problem and then design a representation with the benefit of hind sight. This representation can be used to solve other similar problems more efficiently.

A fact is *weakly irrelevant* to a problem P if there is at least one problem in P's class whose answer can be derived without the fact. We will also say that a fact that is not strongly irrelevant is *weakly relevant* because it is relevant to at least one problem in the class.

Since representations are designed for problem classes, weak irrelevance should not be eliminated and, therefore, our interest is in identifying strong irrelevance.

A direct method for identifying strong irrelevance would be to generate all the problems in the input problem's class, solve them, and then identify statements that were not used in any of these proofs. Clearly this method is impractical.

The method that the representation design system uses abstracts away from the original problem, retaining only its propositional structure. It then identifies a subset of the statements that can not appear in any proof in the problem class based on the connectivity of the problem's *propositional approximation.*

I will show that the facts that can not appear in any propositional proof are strongly irrelevant. I will also show a counter example to the converse, illustrating that the method does not eliminate all strong irrelevance.

3.2.2 An Approximate Strong Irrelevance Filter

The filter begins by constructing a propositional version of the problem by replacing the atomic formulas in each problem statement by propositional symbols, retaining the propositional structure of each statement. Then this version of the problem is converted to clause form. Next, the query fact is extracted from the problem query, converted to propositional clause form, and added to the clause set. Finally, a *connection graph*[Kowalski75] is used to determine which clauses are strongly irrelevant.

The nodes of the connection graph correspond to the propositional clauses and are linked together when they contain literals that resolve.[4] *Any clause that contains a literal that is not connected to any other clause in the graph is strongly irrelevant.*

Disconnected clauses are called *impure* in the connection graph literature and elim-

[4]In a connection graph built from a set of first order clauses, the links between nodes are labeled with the most general unifier of the two literals that resolve.

ination of impure clauses is a standard connection graph technique for eliminating irrelevance. However, using a propositional connection graph to eliminate irrelevance is non-standard and results from our interest in strong irrelevance. Recall that different problems in the same class can have different equality relationships between terms of a sort. The irrelevance elimination procedure must, therefore, allow for equality between any terms. This fact requires us to modify the standard irrelevance elimination procedure. One way to do this is to add axioms for each predicate symbol in a problem so that, given any true fact, we can derive an equivalent true fact by substituting equals for equals. It turns out that once such axioms are included, the same clauses are impure in the first order connection graph as in the propositional connection graph created from the same problem statement (minus the equality axioms).

Having identified the strongly irrelevant clauses, strong irrelevance is eliminated from the original (non-clausal) problem statement by identifying the statements from which those clauses were derived and simplifying those statements (or in some cases removing them). This problem statement is what is used in the representation design process.

Deriving the Propositional Problem

The first step is to extract the propositional structure of the problem. This is done by substituting a predicate's name — as a propositional symbol — for any atomic formula referencing it. For example, $P(x)$ becomes P; $\neg P(x)$ becomes $\neg P$. In doing this, equalities are replaced by the predicate symbol $EQUAL$ and membership expressions are replaced by the symbol $MEMBER$. As this is done, a record is kept of the original statement each propositional statement is derived from. Also, sort statements are removed in this step.

Figure 3.3 gives an example that we will use to illustrate the methods explained in this section. This is the small FAMILIES problem with statements added connecting *married* and *child*. The result of transforming this problem as described so far is shown in Figure 3.4. Note that the query has temporarily been removed.

The next step in this process is to extract a fact from the problem query, convert

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c[married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
Query: find-all $x \mid parent(S, x)$

Figure 3.3: The example problem with the relationship between *child* and *married* added.

$grandchild$
$child \Leftrightarrow EQUAL$
$married$
$child \wedge child \wedge \neg EQUAL \Rightarrow married$
$married \wedge child \Rightarrow child$
$grandchild \Leftrightarrow child \wedge child$
$child \Leftrightarrow parent$

Figure 3.4: The propositional form of the FAMILIES problem with acquired definitions added.

it to propositional form, and add it to the propositional problem statement. The procedures for extracting a fact from each of the query types are as follows:

1. For queries of the form $\Box\ \phi$, the fact extracted is $\neg\phi$. This is exactly what is standardly done in building a connection graph: the statement to be proved is negated.

2. To understand how a fact is extracted from a query of the form $\Diamond\ \phi$, recall that these queries are answered by adding ϕ to a problem situation and looking for a contradiction. This is equivalent to trying to prove $\Box\neg\phi$ and reporting that ϕ is possible unless the proof succeeds. Therefore, ϕ is the fact extracted from $\Diamond\ \phi$.

3. For queries of the form find-all $x \mid \phi$, the fact extracted is also $\neg\phi$ because in order to answer queries of this type the system must construct proofs of ϕ for all individuals that it can.

4. The system can only answer find-the queries when the functions mentioned in them are defined in terms of relations. For example, the query, "find the mother of A" can be answered because *mother-of* is defined in terms of the relations *parent* and *female*. Expanding a find-the query into a fact containing only relations is necessary because the irrelevance filter works with the propositional structure of a problem which does not include functions. The system derives a fact for a query of the form find-the τ by using definitions to expand the formula $\exists x\ x = \tau$ into one that mentions only relations. For example, the system uses the definition of *mother-of* to expand the formula $x = mother\text{-}of(A)$ into

   $\exists x\ parent(x, A) \wedge female(x)$.

   This is the fact extracted from the query, "find-the $mother\text{-}of(A)$.

Extracting the query fact from the problem in Figure 3.4 adds $\neg parent$ to the propositional problem statement.

The next step is to convert the propositional problem statement to clause form, still keeping track of which original problem statement each clause came from.

Before the graph is built, the literals $EQUAL$ and $\neg EQUAL$ are deleted from the clauses. This has the effect of assuming that no clause should be considered impure on the basis of an equality or disequality; this in turn is equivalent to assuming that (dis)equalities are always relevant. This is exactly what we want when reasoning with respect to a problem class because it is usually possible to construct another problem in this one's class in which a (dis)equality is relevant, making $EQUAL$ and $\neg EQUAL$ clauses weakly relevant.

Building the Connection Graph

This process begins by dividing the clauses into two sets. The *query clauses* are those derived from the query fact, the others are called the *descriptive clauses.*

When the graph is complete, all descriptive clauses that do not appear in the graph are marked strongly irrelevant. Then the graph is checked for impure clauses. These are marked strongly irrelevant and deleted from the graph. These deletions may cause other nodes to become impure, in which case these are marked and deleted.

This process continues until no more impure clauses can be found.[5]

As an example, consider Figure 3.5. This is the problem of Figure 3.3 with the following strongly irrelevant statements added:

$\forall x[child(P,x) \Rightarrow haircolor(x, RED)]$
$\forall x[haircolor(x, RED) \Rightarrow pigeon\text{-}toed(x)]$

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P,x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]$
$\forall x \forall y \forall c[married(x,y) \wedge child(x,c) \Rightarrow child(y,c)]$
$\forall x \forall y[grandchild(x,y) \Leftrightarrow \exists z(child(x,z) \wedge child(z,y))]$
$\forall x \forall y[child(x,y) \Leftrightarrow parent(y,x)]$ $\forall x[child(P,x) \Rightarrow haircolor(x, RED)]$
$\forall x[haircolor(x, RED) \Rightarrow pigeon\text{-}toed(x)]$
Query: find-any $x \mid parent(S,x)$

Figure 3.5: A problem used to illustrate the irrelevance filter.

The propositional form of this problem is shown in Figure 3.6. The clausal propositional form is shown in Figure 3.7, while Figure 3.8 shows the connection graph built from the clausal propositional form. The node enclosed in a rectangle in that figure is impure. This node corresponds to the propositional clause, $\neg haircolor \vee pigeontoed$, which, in turn, corresponds to the statement,

$\forall x[haircolor(x, RED) \Rightarrow pigeontoed(x)]$,

in the original problem.

This node is deleted from the graph and the above statement is deleted from the problem. In doing so, the node corresponding to the problem statement

$\forall x[child(P,x) \Rightarrow haircolor(x, RED)]$

becomes impure. Again this node and corresponding statement are deleted.

[5]This is a standard process. see [Kowalski75].

$grandchild$
$child \Leftrightarrow EQUAL$
$married$
$child \wedge child \wedge \neg EQUAL \Rightarrow married$
$married \wedge child \Rightarrow child$
$grandchild \Leftrightarrow child \wedge child$
$child \Leftrightarrow parent$
$child \Rightarrow haircolor$
$haircolor \Rightarrow pigeon\text{-}toed$
$\neg parent$

Figure 3.6: The propositional form of the example problem

$grandchild$
$\neg child, child$
$married$
$\neg child \vee married$

$\neg married \vee \neg child \vee child$
$\neg grandchild \vee child, \quad \neg child \vee grandchild$
$child \Leftrightarrow parent \ \neg child \vee parent, \neg parent \vee child$
$\neg child \vee haircolor$
$\neg haircolor \vee pigeontoed$
$\neg parent$

Figure 3.7: The clausal propositional form of the example problem

Properties of the Irrelevance Filter

We begin by showing that the irrelevance filter is sound, i.e., that it only removes strong irrelevance. It is widely known that impure clauses in the first order connection graph for a problem can not be used in resolution proofs because they can not be used to derive the empty clause. Thus, since any problem is an instance of its own problem class and since impure clauses are irrelevant to the given problem, impure clauses are weakly irrelevant. A clause that is impure in the propositional connection graph for a problem must also be impure in the first order connection graph. Therefore, any impure clause in the propositional graph is at least weakly irrelevant.

The definition of problem class disallows different instances that differ in the propositional symbols they use or in the sense in which they appear. To see this consider the following example. If the clausal form of a problem does not contain the literal $-P$,

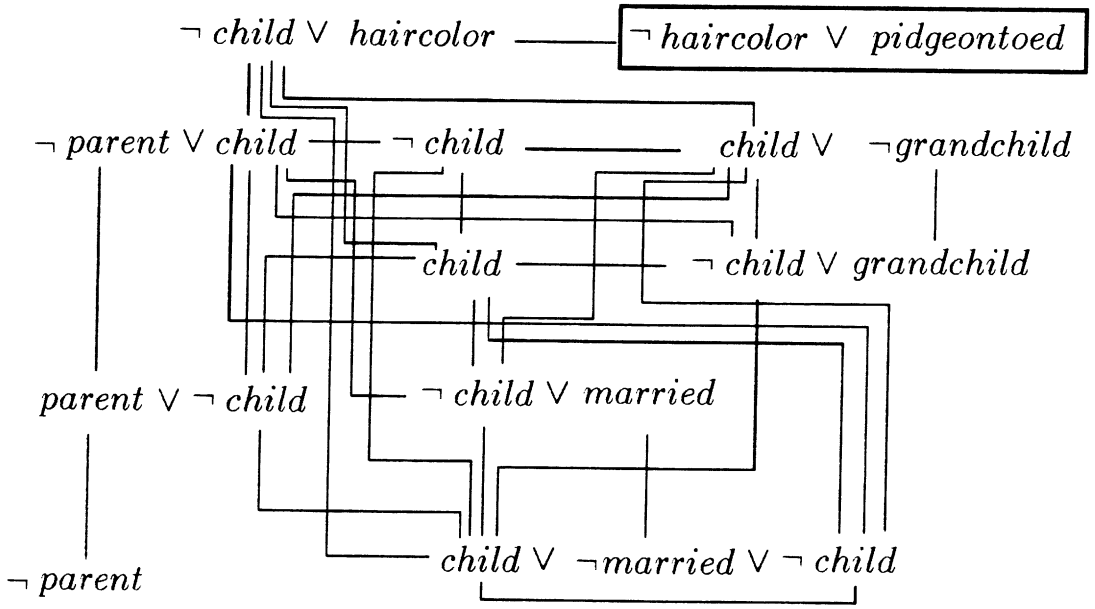


Figure 3.8: The connection graph for the example problem with irrelevance added.

then the clausal form of the problem class also does not. Therefore, no problem in the class contains $\neg P$ in its clausal form. Since the connections in the propositional graph for a problem are between propositional symbols, if a clause is impure in the propositional connection graph for one problem it will be impure in the graph for every problem in the class. Thus, that clause is weakly irrelevant to every problem in the class, i.e., it is strongly irrelevant. Therefore, the filter eliminates only strong irrelevance.

Note that eliminating impure clauses in the first order connection graph without including substitution axioms for every predicate in a problem can eliminate clauses that are not strongly irrelevant. For example, consider the following problem:

$P(a)$
$\forall x[R(g(x)) \Rightarrow Q(x)]$
$\forall x[P(x) \Rightarrow R(f(x))]$
Query: $\Box Q(b)$.

The clausal form of the second general statement is impure in the first order connection graph. However, it is not strongly irrelevant because we can construct another problem in this one's class containing the statement $f(a) = g(b)$.

The irrelevance filter is not complete. In fact, making it so can be very difficult. For

example, if we add the statement

$$\forall x \forall y [f(x) \neq g(y)]$$

to the above problem, then

$$\forall x [P(x) \Rightarrow R(f(x))]$$

is strongly irrelevant.

It seems reasonable for the filter to leave the above statement. However, consider a more blatant problem resulting from definitions. Suppose a problem contains the definition

$$\forall x \forall y [sibling(x, y) \Leftrightarrow \exists p \; child(p, x) \wedge child(p, y) \wedge x \neq y],$$

mentions *child* elsewhere in the problem, but never mentions *sibling*. The propositional version of the problem will contain,

$$sibling \Leftrightarrow child \wedge child \wedge \neg EQUAL,$$

and this will be included in the graph in spite of the fact that it is strongly irrelevant. This is because *sibling* connects to itself through the definition. Concepts like *sibling* above that are irrelevant but are connected through a definition in the graph are called *definitionally irrelevant.* Definitional irrelevance can be detected using the connection graph to identify concepts that are connected to themselves through a definition, but are otherwise disconnected. This is discussed below in section 3.2.4.

Eliminating Strong Irrelevance

Once a collection of strongly irrelevant clauses has been identified, the original problem statement is simplified to remove all mention of them. The desired effect of this process should be the same as converting the problem to clause form and then removing the clauses that mention disconnected literals. However, getting the correct effect on a collection of non-clausal statements is more complicated. For example, suppose the literal P is disconnected and consider the difference between what should be done to the two statements,

$$P \vee Q \Rightarrow R$$
$$P \wedge Q \Rightarrow R$$

The first statement should be simplified to $Q \Rightarrow R$ because either P or Q implies R. However, the second statement can be entirely thrown away because R's truth value

must depend on P.

The insight used in the elimination procedure is that when some literal is disconnected, the solution to the problem will not depend on the truth value of the literal. Therefore, whether it is true or false, the problem solution will be the same. This insight translates rather directly to the following manipulation of a problem statement to eliminate irrelevance. Replace each statement $\mathcal{S}$ in the problem that contains a disconnected literal $\mathcal{L}$ with,

$$\mathcal{S}[\mathcal{L}/true] \vee \mathcal{S}[\mathcal{L}/false].$$ [6]

The new statement is then simplified according to the rules in Figure 3.9.

$$\begin{aligned}
&\neg true \gg false \\
&\neg false \gg true \\
&true \wedge P \gg P \\
&false \wedge P \gg false \\
&true \vee P \gg true \\
&false \vee P \gg P \\
&true \Rightarrow P \gg P \\
&P \Rightarrow true \gg true \\
&false \Rightarrow P \gg true \\
&P \Rightarrow false \gg \neg P \\
&P \Leftrightarrow true \gg P \\
&P \Leftrightarrow false \gg \neg P
\end{aligned}$$

Figure 3.9: Simplification Rules ($\gg$ means "simplifies to").

Continuing the example above, if P is irrelevant, then,

$$P \vee Q \Rightarrow R$$

becomes,

$$[false \vee Q \Rightarrow R] \vee [true \vee Q \Rightarrow R],$$

and then simplifies:

$$[Q \Rightarrow R] \vee false,$$

$$Q \Rightarrow R.$$

Similarly,

$$P \wedge Q \Rightarrow R$$

is transformed as follows,

[6] The notation $\mathcal{S}[\mathcal{L}, true]$ means the statement obtained by replacing $\mathcal{L}$ by $true$ in $\mathcal{S}$.

$[false \wedge Q \Rightarrow R] \vee [true \wedge Q \Rightarrow R]$,
$true \vee [Q \Rightarrow R]$,
$true$.

The second statement simplifying to *true* means that when P is irrelevant, the statement no longer adds a constraint to the problem.

3.2.3 Using the Connection Graph To Identify Definitions

As advertised, the propositional connection graph is used as part of a procedure that finds concepts with equivalent necessary and sufficient conditions. We will call the statements that constitute such conditions the *definition* of the concept. The technique uses the graph to identify a concept and a set of statements in the first order problem that might constitute a definition. Once identified the first order statements are checked to determine if they are, in fact, equivalent necessary and sufficient conditions for the concept.

The analysis of the connection graph finds literals that are connected to themselves through general statements in the first order problem. The technique relies on the following ideas. A connection graph is *pure* when it contains no impure nodes. The set of clauses in which a literal and its negation appear is called the literal's *reference set*. The technique also makes use of a further subdivision of clauses into *general* and *individual* clauses. General clauses do not mention individuals, individual clauses do.

After the graph is built, the general clauses of each literal's reference set are inspected. If they form a pure graph, the first order statements associated with those clauses may constitute a definition of the concept. For instance, consider the propositional clausal form of the definition of *sibling*:

$\neg child \vee EQUAL \vee sibling$
$\neg sibling \vee child$
$\neg sibling \vee \neg EQUAL$.

This set of clauses is a pure graph (see Figure 3.10).

When the general clauses in a literal's reference set satisfy this condition, the system checks the original problem statements from which the clauses were derived to see if they are, in fact, a definition. To see that checking the associated first order statements is necessary consider a simple example that satisfies the above condition

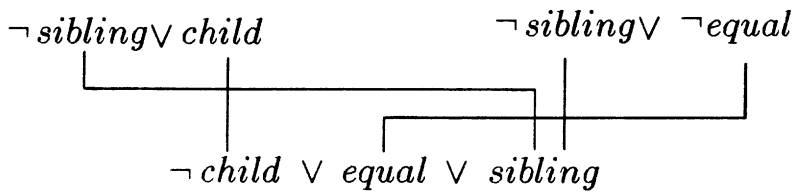


Figure 3.10: Connection graph for the definition of *sibling*.

but which is not a definition. Suppose a concept A has the following general clauses in its reference set:

$A(x,y) \vee \neg B(x,y)$
$\neg A(x,y) \vee B(y,x)$.

These constitute a pure graph, but they are not a definition of A since $B(x,y) \Rightarrow B(y,x)$ follows from these clauses.

A set of statements constitutes a definition for a concept C if, given a statement not involving C, the only new unconditional facts that can be deduced from those statements involve C. For example, from the definition of *sibling* and statements involving *child* and *EQUAL*, we can deduce only unconditional statements involving *sibling*. For instance, given $\neg child(A,B)$ and the definition of *sibling* we can deduce only $\neg sibling(A,B)$. However, the statements

$A(x,y) \vee \neg B(x,y)$
$\neg A(x,y) \vee B(y,x)$

are not a definition of A because from them we can derive $B(x,y) \Rightarrow B(y,x)$.

3.2.4 Removing Definitional Irrelevance

The irrelevance filter discussed above is a procedure that removes irrelevance based on strongly irrelevant clauses in the propositional connection graph. This section discusses using the propositional connection graph to identify strong irrelevance missed by the filter. As discussed in section 3.2.2, a definitionally irrelevant concept can get connected to itself in a graph through its definition. A concept is definitionally irrelevant when it is defined and mentioned only in its definition.

To detect definitional irrelevance, the system first uses the procedure described in the last section to find defined concepts. It then checks the reference set of each defined

concept. If it contains only clauses that are associated with the statements in the concept's definition, the concept is definitionally irrelevant.

3.2.5 Eliminating Irrelevance in Incomplete Problem Statements

As mentioned, there is a trade off between running the filter early, eliminating relevant concepts, and running it later only to find that representations have been designed for irrelevant concepts. Since the cost associated with "resurrecting" relevant concepts eliminated prematurely is much less than designing unnecessary representations, it is best to run the filter early and keep track of what gets eliminated to allow resurrection.

After running the filter, the user is asked about another type of potentially missing information: connections between primitive concepts. The user has already been asked for a definition for every primitive concept. If none was given or if the definition given did not connect the concept, it may be that there is a missing necessary or sufficient condition for the concept that will. To address this possibility, the following heuristic is used:

> "When a literal is determined to be disconnected, ask the user if he can supply necessary or sufficient conditions for it."

This is a heuristic in the sense that it can never be guaranteed to ask all potentially useful questions about the connections between concepts. Any concept could always connect to a non-primitive concept. The only way to augment this acquisition heuristic so that it will ask for all possible connections is to ask about possible necessary or sufficient conditions connecting every pair of concepts in the problem. This is impractical.

When a new statement is provided at this point, it is converted to propositional clause form and added to the connection graph. This may cause statements that were earlier judged to be irrelevant to be resurrected. Also if any totally new concepts are mentioned in a new statement, the representation design system returns to the definition acquisition phase for those concepts and adds any new definitional information into the graph.

3.2.6 Section Summary

Figure 3.11 shows the state of our example problem after definitional knowledge is added. When the irrelevance filter is run on this problem, *married* is identified as irrelevant. As a result, the system asks the user to supply necessary or sufficient conditions connecting *married* and *child* (the other primitive concept in the problem). The user presumably responds with the two statements:

$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow married(x,y)]$
$\forall x \forall y \forall c[married(x,y) \wedge child(x,c) \Rightarrow child(y,c)]$.

In this case, running the irrelevance filter does not change the set of represented concepts identified in the last step. It remains as $\{child, married\}$.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q,S)$
$\forall x\ child(P,x) \Leftrightarrow x = R$
$married(Q,P)$
$\forall x \forall y[grandchild(x,y) \Leftrightarrow \exists z(child(x,z) \wedge child(z,y))]$
$\forall x \forall y[child(x,y) \Leftrightarrow parent(y,x)]$
Query: find-all $x \mid parent(S,x)$

Figure 3.11: The example problem with definitions added.

3.3 Including Concepts With Restrictions

The system now adds to the set of represented concepts those that have explicit restrictions on them. It looks for mixed statements that it recognizes as restrictions and adds the concepts in those statements. It recognizes four forms of mixed statements as restrictions:

1. A statement that is an atomic formula with a negation at the top level, e.g., $\forall x \neg brother(M,x)$.

2. A conditional statement containing one or more equalities between universally quantified variables and constants; for example, $\forall x[child(P,x) \Leftrightarrow x = R]$.

This procedure will not identify concepts that have implicit mixed constraints on them. For example, it will not identify the restriction on *brother* implied by the statements

$\forall x \neg sibling(M, x)$
$\forall x \forall y [brother(x, y) \Leftrightarrow sibling(x, y) \wedge male(y)]$,
i.e., $\forall x \neg brother(M, x)$.

3.4 Completing a Problem's Sort Signature

The steps described in the last few sections identify the set of represented concepts for a problem. The system defines a representation for each represented concept directly from its sort signature. For example, given that *married* is a represented concept, the system includes

```
married: relation(family-member,family-member)
```

in the initial representation description.

Generally, analytical reasoning problems contain only partial sort information which the system completes using a few simple techniques. In cases where these techniques do not suffice in determining needed sort information, the system asks the user.

A few restrictions are placed on problem statements to make it possible to extract sort information. An initial problem statement must make reference to one or more domain sorts that are assumed to be sorts of domain individuals. For example, there is one domain sort in FAMILIES which is referred to as *family-member*. All domain sorts are assumed to be disjoint from all others.

All constants in the problem statement must have their sorts declared. For example, note that in the FAMILIES problem there is a sort declaration (e.g., N : *family-member*) for every individual mentioned.

To describe the fact that *family-member* is a domain sort, the system declares it as follows:

```
(deftype family-member specializes individual disjoint)
```

This statement means that family members are a subtype of the individuals in the "world."

Given sorts for all the individuals, the system attempts to determine the domain of relations and the domain and range of functions by inspecting the statements in which they appear.

It attempts to determine the domain of a relation in two ways. When there is a problem statement that relates individuals, the domain of the relation involved is defined from the sorts of the individuals. For example, from the statements

$A : family\text{-}member$
$B : family\text{-}member$
$married(A, B)$

it is determined that *married* is a binary relation on family members.

When the domain of a relation can not be determined directly, the system looks for statements that use variables in the argument position in question. It then tries to determine the sort of that variable by its other uses in the statement. For example, given,

`sibling: relation(family-member,family-member),`

and the statement,

$\forall x \forall y [brother(x, y) \Leftrightarrow sibling(x, y) \wedge male(y)]$,

the representation design system can determine that *brother* is a binary relation defined over family members and that *male* is a unary relation over family members.

The system attempts to determine the sorts in the domain of a function in the same way as relations. For the range of a function, it looks for uses of the function as an argument in a relation whose domain is known. If this fails to produce a sort for the range, then the system looks for an equality between an application of the function and another term whose sort is known. For example, from the statements,

$A : family\text{-}member$
$B : family\text{-}member$
$father(B) = A$,

father is defined as

`father: function(family-member,family-member).`

Problem statements are also checked for consistent use of functions and relations. When the sort of a function or relation is ambiguous in the problem statement, the system asks the user to disambiguate.

3.4.1 Section Summary

The primitive concepts of our example problem are *married* and *child*. Because it appears in the find-all query, *parents* is also included. The representations of these concepts are defined as follows:

```
married: relation(family-member,family-member)
child: relation(family-member,family-member)
parent: relation(family-member,family-member).
```

3.5 Chapter Summary

This chapter has described how an initial representation is derived from a problem statement. The basic strategy is to identify a collection of concepts that is sufficient for representing the problem and then to define representations for them. It is desirable to identify the smallest collection of concepts that is sufficient for this purpose to avoid designing redundant representation machinery. The smallest collection of concepts that is sufficient is the collection of relevant primitive concepts. Although not logically necessary, the system also includes in this set defined concepts that have relevant restrictions on them and defined concepts appearing in find-all queries.

The relevant primitive concepts are identified in three steps:

1. The primitive concepts are identified and the user is given the opportunity to give definitions for any of these that are not, in fact, primitive.

2. Irrelevant concepts are eliminated from the problem. A key idea in understanding irrelevance is the notion of strong irrelevance: a fact is strongly irrelevant to a problem class when it can not be used in solving any problem in the class. Since representations are designed for problem classes this is the kind of irrelevance we attempt to eliminate.

3. Concepts that have explicit restrictions on them are identified. Some concepts that have implicit restrictions on them (i.e., those for which a restriction follows from the problem but is not stated) are identified later in representation design.

4. Concepts appearing in find-all queries are added to the set of represented concepts.

The final step in deriving the description is to define representations for the concepts identified by the previous steps. The initial representations for the concepts identified in the steps above are isomorphic to the sort signature of those concepts. Analytical reasoning problem do not include complete sort information, but usually sort signatures can be derived from the information given by a few simple techniques. When this is not possible, the system asks the user for the missing sort information.

The resultant representation is complete but does not capture any of the constraints of the problem statement. This means that the representation can be used to build problem situations in which, for example, *married*, *child*, and *parent* are arbitrary relations over *family-member*. For instance, we can create a situation in which A is the child of B but B is not the parent of A. The rest of the processes of representation design attempt to maximally constrain this representation while preserving completeness.

Chapter 4

Classification

We have defined a representation to be a mapping from concepts to structures that enforce constraints. The representation design system captures constraints on a concept by representing it in terms of a library structure enforcing those constraints. Part of the system's library of structures is shown in Figure 4.1. Concepts are represented by instances of library structures. For example, *married* is represented as an instance of `relation`:

`married: relation(family-member,family-member).`

The system tries to capture as many of a concept's constraints as possible by representing it with the most specialized structure (or structures) available, i.e., the available structures that together enforce the most constraints on the concept without enforcing constraints that are not true of the concept.

The most specialized structures for representing a concept are identified by classifying the concept in a hierarchy of the library structures (again see Figure 4.1). The definitions of the property names appearing in Figure 4.1 are given in Figure 4.2. If we ignore the node labels, the figure depicts a standard hierarchy: The nodes represent classes of concepts, with more specialized classes (containing concepts with more properties) appearing below less specialized classes. For example, the node appearing below `bin-rel` on the link labeled "symmetric" denotes the class of symmetric binary relations, a specialization of the class of binary relations.

Some nodes in the hierarchy have several unlabeled links coming into them. These denote specialized classes of concepts that are the intersections of the classes linked

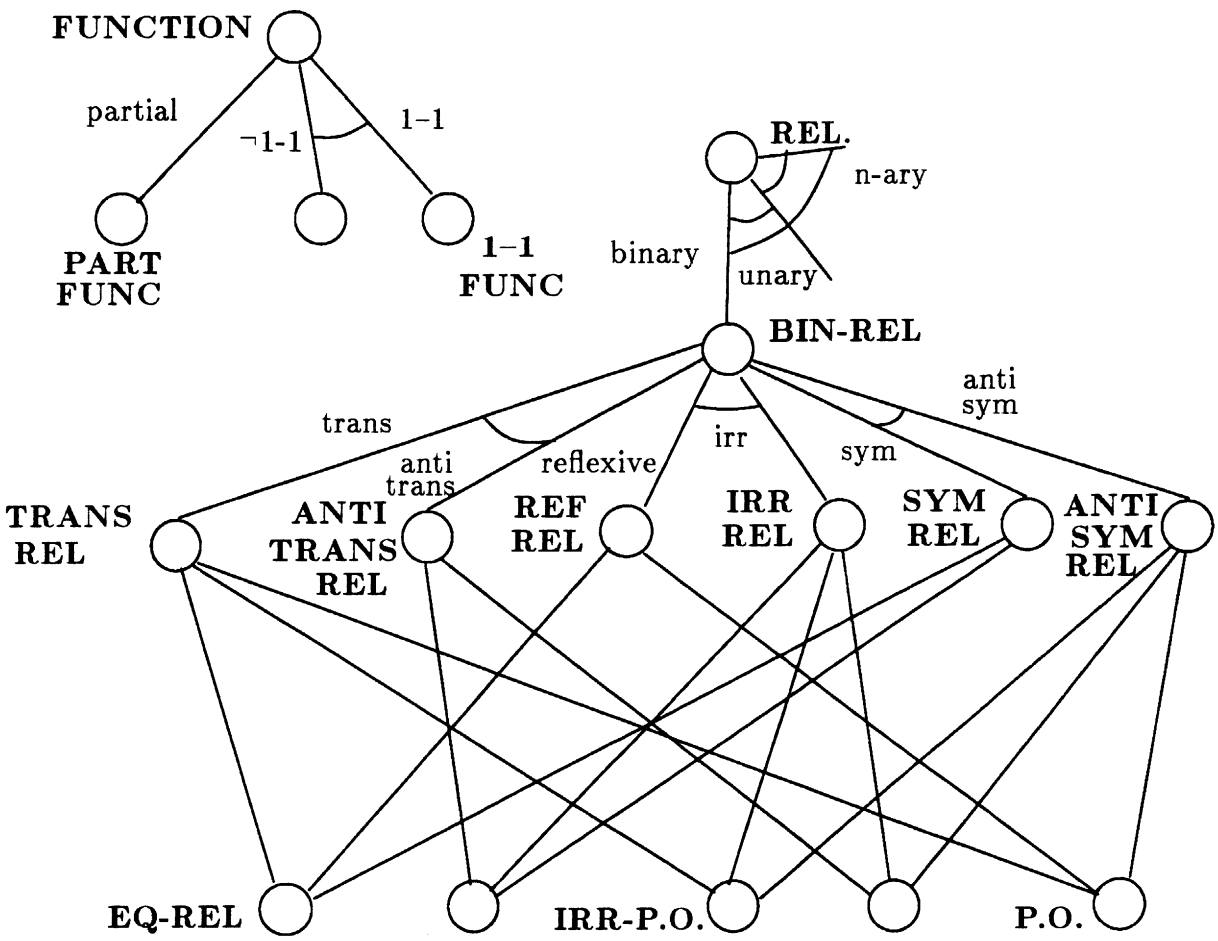


Figure 4.1: Part of the structure library.

Property	*Definition*
reflexive	$\forall x\ R(x,x)$
irreflexive	$\forall x\ \neg R(x,x)$
symmetric	$\forall x \forall y [R(x,y) \Rightarrow R(y,x)]$
antisymmetric	$\forall x \forall y [R(x,y) \wedge x \neq y \Rightarrow \neg R(y,x)]$
transitive	$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow R(x,z)]$
antitrans	$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow \neg R(x,z)]$
total	$\forall x \exists y\ y = f(x)$
1-1	$\forall x \forall y [f(x) = f(y) \Rightarrow x = y]$

Figure 4.2: The definitions of the properties labeling links in Figure 4.1.
(Properties not defined here are determined by inspection.)

from above.

Nodes are labeled with library structures. The structure labeling a node captures the properties of the node's class. Placing a concept in a class labeled by a structure is how the system identifies that structure for representing the concept. For example, the node labeled **sym-rel** denotes the class of symmetric binary relations; the library structure **sym-rel** enforces symmetry. By classifying a relation as symmetric, the system identifies a structure for representing that relation which captures symmetry.

Nodes are included in the hierarchy for one of three reasons. We have already explained the first reason: a node is included for a class when the system knows a specialized structure for representing concepts in the class. Representations containing specialized structures are better because fewer situations can be expressed in them. Therefore, a problem solver using a specialized representation explores a smaller search space. For example, when the concept F is represented as a **function**, problem situations can be expressed that have multiple domain elements mapping to the same range element. When F is represented as a **1-1 function**, it is more constrained because the legal situations in the new representation are a subset of those in the previous representation, i.e., those situations in which domain elements map to unique range elements.

The second reason that nodes are included in the hierarchy is that some structures are included in the library because they capture a combination of constraints. This applies to nodes denoting the intersection of classes. Structures capturing a combina-

tion of constraints are included in the library only when they exploit the combination to gain efficiency. For instance, the class denoting equivalence relations is included because there is a library structure **eq-rel** which captures the combination of reflexivity, symmetry, and transitivity. **Eq-rel** is included in the library because it captures these constraints more efficiently than the combination of **ref-rel**, **sym-rel**, and **trans-rel**. A more in depth argument for when to include structures in the library is presented in section 4.2.

The third reason that nodes are included in the hierarchy has to do with concept introduction: An unlabeled node is only included when the system has rules for introducing a new concept allowing it to represent an existing concept differently. This is discussed in Chapter 5.

The system also designs representations for sorts. While concepts are represented as instances, sorts are represented as data types which are defined as subtypes of library structures. For example, the representation of *family-member* is defined as

(deftype family-member specializes individual disjoint),

where **individual** is the library structure for representing domain individuals. Just as the sort *family-member* is a subsort of the problem domain, the type **family-member** is defined to be a subtype of **individual**. Instances of **family-member** are used to represent individuals like N from the FAMILIES problem.

The system treats sorts in a special way, classifying them in a separate hierarchy (shown in Figure 4.3). Classifying sorts increases the kinds of constraints that can be captured by classification. The structures in the sort hierarchy enforce constraints common to all individuals of a sort as well as constraints between individuals of a sort. Axiom schemas giving the meaning of each of the constraint names used in Figure 4.3 are given in Figure 4.4

One way to view classification is as a process of identifying useful constraints in a problem. This gives the system a way of directing a search for interesting constraints, saying that they are those that library structures can capture. Without such a technique the representation design process is faced with an unstructured collection of statements, not knowing which are important for design or which it should try to capture first.

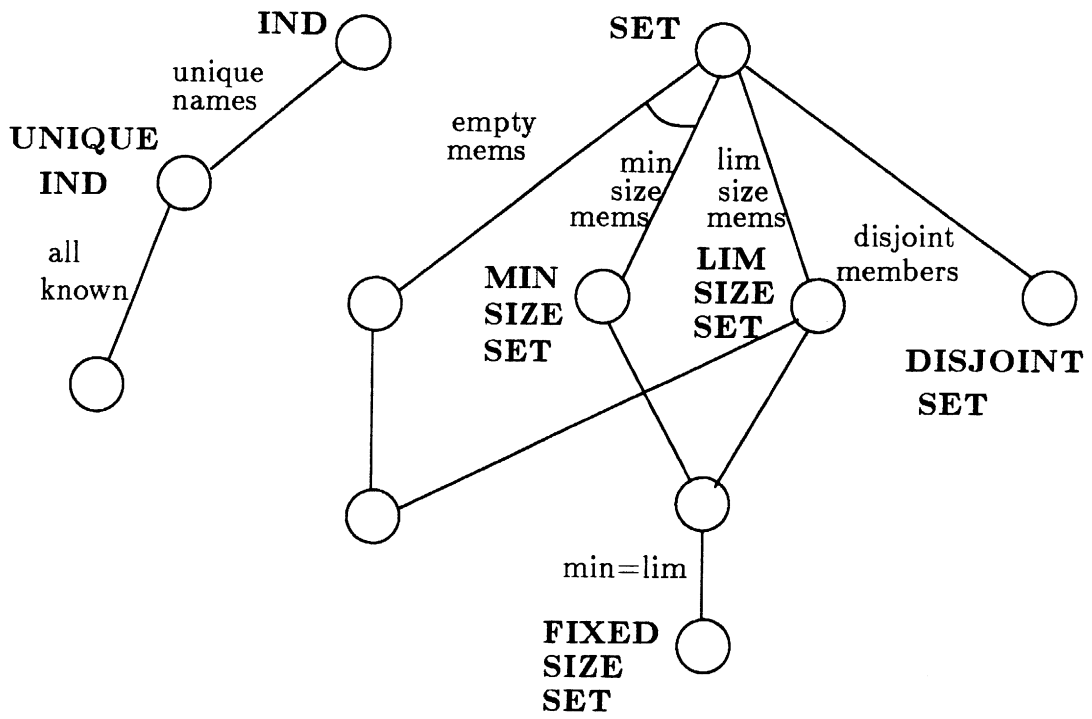


Figure 4.3: The hierarchy used to classify sorts.

minimum size	$\forall x : S \exists y_1, \ldots, y_n [\{y_1, \ldots, y_n\} \subseteq x]$
limited size	$\forall x : S \exists y_1, \ldots, y_n [x \subseteq \{y_1, \ldots, y_n\}]$
empty	$\exists x : S[x = \emptyset]$
disjoint	$\forall x : S \forall y : S \forall z [z \in x \wedge z \in y \Rightarrow x = y]$

(The symbol S in these schemas is the sort being classified.)

Figure 4.4: Axiom schemas defining the meaning of the constraint names used in the sort hierarchy.

Classification can be used to assist in the acquisition of missing information. The constraints that it asks about are assumed to be important enough in representation design and applicable to a sufficiently wide variety of problems that they are worth asking about when they are not found in a problem statement. This assumption is discussed below in section 4.2.

Classification can not capture all possible kinds of constraints. Intuitively, we can divide possible constraints into those that constrain a concept in terms of itself and those that constrain a concept in terms of other concepts. An example of the first type is,

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)]$$

and an example of the second type is,

$$\forall x \forall y \forall z [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \Rightarrow married(y, z)],$$

i.e., "two different parents of an individual are married."

Classification can only capture constraints of the first kind because it classifies concepts individually. Similarly, sort classification can only capture constraints between individuals in a sort. For example, one sort introduced during design of a representation for FAMILIES is *parent-set*. Classification can capture the constraint, "parent sets are disjoint from each other," which is a constraint between individuals in the sort. It can not capture a constraint like, "every parent set is a couple," which is a constraint between sorts.[1]

Operationalization can capture both types of constraints. This is one reason it is done after classification.

4.1 The Classification Process

Before classification begins, all problem statements are converted to *implication normal form*. This is the following

[1]Note, however, the system captures taxonomic relationships between sorts as it builds up a sort structure during design. This is described in section 5.1.1.

$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m))$,

where each of the ϕ_i and ψ_i are literals. Note that this is equivalent to

$(\phi_1 \wedge \cdots \wedge \phi_n) \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)$

and we shall often use this latter form in our examples even though the system is actually using the former.

The main loop in classification systematically selects concepts included in the initial representation and tries to identify properties in an effort to "push" down into the concept hierarchy. It uses a concept's definition to determine where to begin. For example, classification of *married* begins at the **relation** node.

When classification reaches a node in the hierarchy, the system looks at the properties labeling links leaving that node and tries to determine which properties the concept has. For example, the classification of *married* begins with the system looking at the links leaving the **relation** node, trying to determine if *married* is unary, binary, or n-ary.

The hierarchy is also annotated to show which properties leaving a node are disjoint (indicated by the arcs connecting links in Figure 4.1 and Figure 4.3). For example, unary is annotated as disjoint from binary. The system tries to identify one property from each group of disjoint properties leaving a node. For example, when classifying a relation at the **bin-rel** node, the system tries to determine whether the relation is transitive or antitransitive, reflexive or irreflexive, etc. At the same time, the system tries to find counter examples to properties labeling each of the links.

When the system is unable to identify one property from a disjoint group or to demonstrate a counter example for each property in the group, it asks the user about the properties. For example, if the system is unable to determine whether a binary relation is transitive, antitransitive, or to find counter examples for both, it asks the user about each property in turn.

Classification at a node is complete when the system has examined each disjoint group of properties at that node and either it has identified one property from the group or it has determined that the concept being classified has none of the properties in that group. Classification then continues with each of the nodes below the properties identified. For example, after *married* is classified as binary, classification

continues with the node labeled **bin-rel**; when the system determines that *married* is irreflexive, symmetric, and antitransitive, classification completes at this node.

Classification handles unlabeled links differently. When classification completes at a node with unlabeled links leaving it, those links are marked traversable. Classification does not proceed down an unlabeled link until all other links entering the node below are marked traversable. For example, classification will only proceed from the node labeled **sym-rel** to **eq-rel** when the links leaving the nodes labeled **sym-rel**, **ref-rel**, and **trans-rel** have all been marked traversable.

When classification of a concept reaches a node that has a library structure associated with it, the representation of the concept is redefined in terms of that library structure. For example, when *married* is classified as irreflexive, its representation is redefined as

```
married: irr-rel(family-member,family-member).
```

A concept can be placed in multiple classes, each of which may be labeled. When this occurs the system defines a new data type to represent the concept. The new data type is a mixture of the data types labeling each node. For example, if a relation is classified as symmetric and reflexive but is neither transitive nor antitransitive, the system defines a new data type to represent that relation. The new data type is a mixture of **ref-rel** and **sym-rel**. All of the library structures have been defined so that they are mixable in this fashion. This is standard practice in object oriented programming so the details are omitted here. We will use the convention of naming data types that are mixtures by appending the names of the component data types. For example, the data type that is a mixture of **ref-rel** and **sym-rel** will be named **ref-sym-rel** (which is equivalent to **sym-ref-rel**).

Sorts are classified by the algorithm described above, operating instead in the sort hierarchy. The sorts that a concept is defined over are classified before that concept is classified. The reason for this has to do with concept introduction. As will be explained in Chapter 5, new sorts can get introduced during the classification of existing sorts. Often when a new sort is introduced, the concepts defined over an existing sort get replaced by new concepts. In this case, any effort expended classifying the existing concepts is wasted. The converse situation does not occur, i.e.,

the classification of a concept may introduce new concepts but never introduces sorts that replace existing sorts. Therefore, the system waits to classify a concept until after its sorts have been classified.

4.1.1 Answering Questions Posed by Classification

A "brute force" approach to answering questions posed by classification would be to construct a statement of the property in question and then to use a theorem prover to try to show that the statement follows from the problem statement. For example, to determine whether *married* is symmetric, we could construct the statement,

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)],$$

and then instruct a theorem prover to determine whether this statement follows from the problem.

However, recall that analytical reasoning problems are usually missing information. This means that a problem statement may not contain information about a property, either because it is not relevant to solving the problem or because the information is simply missing. This is problematic for the brute force approach. If we turn a complete theorem prover loose on some proof attempt, it may never halt and we can never know whether this is because we have not given it enough time or the problem statement is incomplete.

The representation design system constructs a statement of the property and then uses a semi-decision procedure to try to prove that it follows. The semi-decision procedure is implemented as a combination of three mechanisms: a rewrite system that simplifies problem statements, a matcher that recognizes some syntactic variations in the patterns it is matching, and mechanism that attempts to find a counter example to a general property that the system is looking for. When this procedure halts and reports a failure to determine the status of some property, the system assumes that the information is missing from the problem and asks the user about it.

The advantage of this approach is that it avoids asking the user most questions that would be considered obvious, and at the same time avoids getting bogged down trying to prove that some constraint holds when a problem statement may not contain the

information.

The approach is implemented as follows. First, as problem statements are added to the system's description, they are simplified by the rewrite system. The intention of this step is to partially "canonicalize" statements, i.e., rewrite them so they will be recognized by classification.

The rewrite system is implemented in the usual way, as a collection of rules that match and replace patterns in statements (see appendix A for a more in depth discussion of the rewrite system). The current body of rules mainly exploits properties of sets to simplify statements. One of the rules is paraphrased as follows:

> If a statement is a conjunction of expressions, two of which are of the form $x \in S$ and $y \in S$ and a third expression in the conjunction has the form $x \neq y$, then replace the first two expressions by $\{x, y\} \subseteq S$.

This rule will, for example, rewrite the statement

$$\forall x \exists y \exists z [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z]$$

as

$$\forall x \exists y \exists z [\{y, z\} \subseteq parents(x) \wedge y \neq z].$$

Both statements express the fact that every individual has at least two parents, but classification will recognize only the second statement as the "min size" constraint (Figure 4.4) on members of the sort *parent-set* (sets of parents of the same individual).

To check the problem for a property, the system generates a statement expressing that property from the constraint axiom schema by substituting the appropriate concept for the concept symbol in the schema. All properties in the hierarchies have schemas associated with them (unless they can be checked by inspection). Figure 4.2 gives the schemas for properties in the concept hierarchy.

The schemas for properties in the sort hierarchy are given in Figure 4.4. A first order statement is constructed from these schemas in two steps. First, the name of the sort being classified is substituted for the sort symbol in the schema. Second, when the sort was defined by the system as a sort for the range elements of a function, each sorted variable in the schema is replaced by an application of that function. For example, recall that the axiom schema for the minimum size constraint is

$\forall x : S \exists y_1, \ldots, y_n [\{y_1, \ldots, y_n\} \subseteq x]$.

The sort *parent-set* is defined by the system when the function *parents* is introduced. When classifying this sort, S is replaced by *parent-set* and all occurrences of x are replaced by $parents(x)$, yielding

$\forall x : parent\text{-}set \exists y_1, \ldots, y_n [\{y_1, \ldots, y_n\} \subseteq parents(x)]$.

When this statement is matched against problem statements, $y_1, \ldots, y_n$ will match any sequence of variables, allowing the system to check for a constant set of any size.

The matcher allows statements that are syntactic variations of each other to match. One type of variation that it handles occur with commutative, associative connectives. For example, the matcher treats $\wedge$ and $\vee$ as commutative, associative connectives so that two statements will match if changing the order of a group of conjuncts in one of the statements makes it unify with the other statement. For example, the following two statements match:

$\forall x \forall y [P(x,y) \wedge (Q(x,y) \wedge R(x,y))]$
$\forall u \forall v [(R(u,v) \wedge P(u,v)) \wedge Q(u,v)]$.

4.1.2 An Example of Classification

We demonstrate the classification of *married* as it is performed by the system when given the problem used in our running summary. The purpose of this example is to illustrate the classification process and the knowledge acquisition behavior that the system exhibits. The problem statement modified by processes of Chapter 3 is shown in Figure 4.5. Note that the system has converted these statements to implication normal form. However, in our examples, we will leave statements in their original form so long as their translation to implication normal form is straightforward.

Before *married* can be classified, *family-member* must be classified in the sort hierarchy. Since *family-member* is represented as

`(deftype family-member specializes individual)`,

classification begins with the `individual` node in Figure 4.3. The first question is whether the names mentioned in the problem statement refer to unique individuals. To determine this, the system looks for disequalities between all the individuals. The

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c[married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
Query: find-all x: $parent(S, x)$

Figure 4.5: State of the example problem at the end of the Chapter 3.

problem statement does not contain any, so the system asks the following question:

Are P, Q, R, and S all different individuals?
Yes.

All the analytical reasoning problems I have studied make the unstated assumption that individuals with different names are different. However, this could easily not be the case, so the system explicitly asks about this assumption. Since, in this case, the answer is "yes," **family-member** is specialized as

```
(deftype family-member specializes unique-individual).
```

The structure **unique-individual** represents a sort of individuals with the property that individuals with different names are different. This is implemented by an equality procedure associated with **unique-individual** that simply compares individual's names.

The next question posed by classification is whether the problem statement mentions all the individuals of sort *family-member*. The system knows no way of checking the problem statement to see whether this is true, so it asks the question

Are P, Q, R, and S all the family members?
Don't Know.

Note that, as in this example, the system allows the user to answer questions with "don't know" when determining the answer requires solving the problem, e.g., one can not tell whether or not all the family members are known until very late in the process of solving the FAMILIES problem. The system always treats such an answer as a "no" response.

Classification of *family-member* now terminates. The system proceeds with the classification of *married*. It is classified as binary by inspecting its representation. Next the system tries to determine whether it is reflexive or irreflexive. It looks for statements of these properties in the problem, and looks for counter examples. None are found so the user is queried:

Is MARRIED reflexive? No.
Is MARRIED irreflexive? Yes.

In response to this answer, **married** is specialized as

married: irr-rel(family-member,family-member)

and the statement $\forall x \neg married(x, x)$ is added to the problem.

Then the system turns to the question of whether *married* is symmetric or antisymmetric. Again the user is queried:

Is MARRIED symmetric? Yes.

In response to this answer, **married** is further specialized as[2]

married: sym-irr-rel(family-member,family-member)

and the following statement is added to the problem:

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)].$$

Classification now proceeds to the question of transitivity and, again, the problem statement provides no assistance, so the user is asked:

Is MARRIED transitive? No.
Is it antitransitive? Yes.

Married is now specialized as

married: antitrans-sym-irr-rel(family-member,family-member)

and the following statement is added to the problem,

$$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)].$$

This classification effort is now complete; it has reached the leaf node denoting irreflexive, symmetric, antitransitive relations (shown shaded in Figure 4.6). Since there is no structure associated with this node, the representation of *married* remains as shown above. However, as we will see in Chapter 5, this node is treated

[2]Recall that **sym-irr-rel** is actually a mixture of the structures **sym-rel** and **irr-rel**.

specially by concept introduction. The problem statement resulting from this classification is shown in Figure 4.7 with the added statements enclosed in a box.

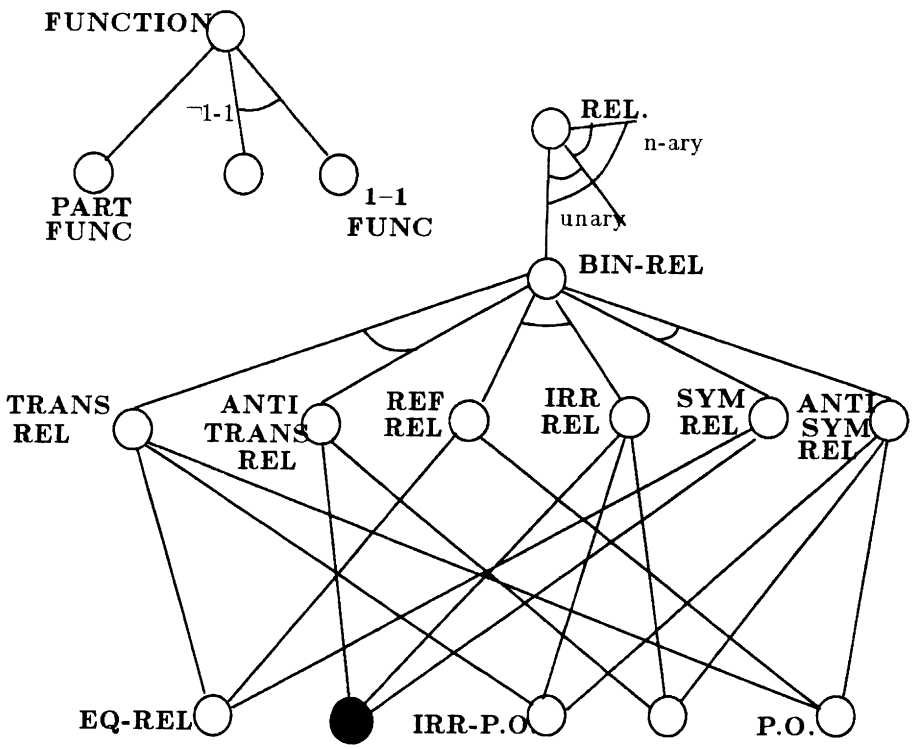


Figure 4.6: The node reached in classifying *married*.

The specialized representation designed so far is:

```
(deftype family-member specializes unique-individual)
married: antitrans-sym-irr-rel(family-member,family-member)
child: relation(family-member,family-member)
parent: relation(family-member,family-member).
```

4.2 Assumptions About Library Structures

For the library to be useful, the properties found in it should appear in a wide variety of problems and enforcing them as constraints should provide significant leverage in problem solving. In this case, classification becomes a technique for recognizing when a problem contains properties that the representation design system knows efficient

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y [child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$

$\forall x \neg married(x, x)$
$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$
$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$

Query: find-all x: $parent(S, x)$

Figure 4.7: The small FAMILIES problem after *married* has been classified.

ways of capturing. It assumes that certain properties are worth looking for in a problem because of the leverage obtained in specialized representations that enforce those constraints. For example, it assumes that it is worth trying to determine when the sets of the same sort are disjoint from each other because the system has efficient ways of capturing disjointness.

There is at least one rule of thumb in searching for structures that are widely applicable and provide significant leverage: the more general the properties that a structure captures the wider its applicability. However, it is usually the case that the more general a structure is the less leverage it provides in representation design. One of the challenges in finding the "right" collection of structures to look for is trading off generality against usefulness.

In this research, useful structures were found by studying the representations that people use to solve analytical reasoning problems. I looked for structures that people commonly use in their representations and I identified the properties that those structures capture. The current library population is the result of this investigation conducted with twenty analytical reasoning problems. This investigation was discussed in section 1.5.1.

In retrospect, the structures that people use turn out to be constraints (or combinations of constraints) having particularly efficient implementations. Consider an

example of how much leverage the library structures afford because of this. One specialization of **set** in the sort hierarchy is **disjoint-set**. **Set** provides an equality procedure for sets of the same sort which reports that two sets are equal if and only if all elements of both sets are known and they are the same. **Disjoint-set** provides a more efficient equality procedure that exploits the additional constraint that the sets of a sort are disjoint: it reports true if it can show that two sets share any members. Thus, it is more efficient in two ways: it does not necessarily have to check all members of the sets its comparing and it can work even if some members of those sets are unknown.

It may be that the particular collection of structures in the current library prove to be less applicable as different problems are investigated. However, notice that the structures in the library are similar to concepts that have been important in mathematics for a long time. I did not build the library by attempting to replicate what mathematicians consider important. Instead, I tried to capture what I found people using in representations. The fact that I ended up with a collection of structures similar to those that mathematicians consider important reinforces the likelihood that these have fairly broad applicability.

The issue of whether the current collection of structures will provide significant leverage in a wide variety of problems is empirical. So far, the only substantive claim I can make is that they do provide significant leverage in designing representations for twenty analytical reasoning problems.

Assuming that the library structures provide significant leverage and are widely applicable, classification can be viewed as the following useful knowledge acquisition heuristic:

> "Properties of library structure are useful enough that if they can not be identified in a problem, they are worth asking about."

This is only a heuristic because it can cause the system to acquire information that is irrelevant to solving a problem and this, in turn, can cause it to over-design a representation.

4.3 Capture Verification

The system designs representations for concepts in which constraints on those concepts are captured. An important consequence of a constraint being captured in a representation is that it is guaranteed to be true in any situation created in that representation. In fact, the following stronger statement is also true: a constraint is captured in a representation if and only if it is true in every situation that can be created with that representation. This section shows how this fact has been turned into a test, enabling the system to use a representation directly to determine whether a statement is captured in it. This test is called *capture verification.* The section also explains why such statements can be removed from the system's consideration.

4.3.1 How Capture Verification Works

The basic idea of capture verification is to prove that a statement is captured in a representation by demonstrating that it is true in every situation that can be created in that representation. This is a constructive proof technique: A representation is used to build situations, i.e., collections of data structures, and then those situations are inspected to determine if the consequences of the statement are true. The reason this test is useful is that the system does not really have to check *every* possible situation to verify that a statement is captured. In fact, it only has to check one "minimal" situation. This is guaranteed by requirements imposed on library types which are described in section 4.3.3.

Consider an example of capture verification: Suppose the relation *couple* defines a sort whose members are married couples (i.e., $couple(x)$ is true if x is a set of size two whose elements are family members married to each other). Suppose further that this sort, call it *married-couple*, has been classified as having fixed sized disjoint members and, consequently, is represented as

```
(deftype married-couple specializes
fixed-size-disjoint-set(2,family-member)).
```

Finally suppose a problem containing *couple* and *married-couple* also contains the statement

$$\forall x \forall y \forall z [couple(\{x, y\}) \wedge x \neq y \wedge couple(\{x, z\}) \wedge x \neq z \Rightarrow y = z].$$

Capture verification proves that this statement is captured by constructing a situation representing the antecedent of the statement and then checking that situation to see if the consequent is true. First it creates two anonymous family members by instantiating the **family-member** ADT without giving names for the instances. In this exposition we will call these instances **x** and **y**. After creating these, the system assumes that they are disequal by adding the statement $x \neq y$ to the situation. An operation corresponding to adding this disequality is supported by an equality system (discussed in Chapter 7) that is part of all specialized representations.

Next the system creates an instance of **married-couple** whose elements are **x** and **y**. It then performs similar steps to create **z**, assume that $x \neq z$, and create a married couple containing **x** and **z**. When the second instance of **married-couple** is created, a procedure associated with **married-couple** enforces the disjointness property by combining the two separate instances (since they share **x** and, therefore, must be the same). Then another procedure enforces the fixed size of members of *married-couple* by combining the instances **y** and **z** (this is true because neither **y** nor **z** is equal to **x** and because the **married-couple** may contain only two individuals).

When capture verification checks the situation created, it finds the consequent of the above statement to be true and concludes that the statement is captured.

In general, capture verification can directly check statements in implication normal form by creating a situation representing the conjunction of antecedents and then checking that all the consequents are true in that situation.

A conjunction of atomic formulas is considered to be in this form, i.e., n can be 0. To test a general statement in this form, capture verification creates anonymous individuals for each universally quantified variable mentioned in the statement and then tests whether the statement is true in the situation containing those individuals. For example, suppose that *child-set* is a sort whose members are sets of individuals that are children of the same couple and that *child-set-of* is a function mapping an individual to the set of children he/she is a member of, i.e.,

$$\forall x \forall y [x \in \textit{child-set-of}(y) \Leftrightarrow \exists z (\textit{child}(z, x) \wedge \textit{child}(z, y))].$$

Further suppose that we have designed representations **child-set-of** and **child-set**. Then the statement,

$\forall x[x \in \textit{child-set-of}(x)]$,

is tested as follows. First we create an anonymous individual `x` and then we create an instance of `child-set`, call it `y`. Next, we make `y` be the image of `x` under `child-set-of`. Finally, we check to see if `x` is a member of `y`.

4.3.2 How Capture Verification is Used

As representations are designed, constraints expressed in problem statements get captured by those representations. Capture verification is used to identify statements in a problem that express captured constraints. It is run whenever classification of a concept terminates. The system removes statements identified by capture verification so that the problem statement is, at any point, an accurate record of which statements are not captured by a representation.

The problem statement plays two important roles as a record of statements left to be captured. First, when a problem statement is empty, the system knows that all of the constraints of a problem are captured and, consequently, the design effort is complete. Second, concept introduction creates alternative problem formulations which are compared. The comparison process relies on the alternative formulations of the problem statement being accurate records of which statements are uncaptured in each.

4.3.3 Why Capture Verification Works

This section answers two questions: Why is is sufficient for capture verification to create and check only minimal situations? And why is it sufficient for it to check only *one* minimal situation? The reason that only minimal situations must be checked is that library structures are required to be *monotonic*. When a structure is monotonic, specific facts that are true in a situation created with that structure remain true no matter what information is subsequently added.

As an example of a structure that violates monotonicity, suppose `relation` was implemented as one list of n-tuples with the absence of some n-tuple being interpreted as negation, e.g., the absence of $< A, B >$ from the `married` list meaning

$\neg married(A, B)$. In this implementation **relation** is non-monotonic because adding the pair $< A, B >$ changes the truth value of $married(A, B)$.

Capture verification does not have to check "larger" situations because monotonicity guarantees that such situations do not change the truth value of the facts in a minimal situation.

One reason that capture verification need only check *one* minimal situation is that all library structures are required to be *logically sound.* When a structure is logically sound, *every* consequence of its invariant axioms is true in situations created with the structure. For example, **sym-rel** is sound because every logical consequence of the symmetry axiom is true in situations created with it. In particular, if R is represented as a symmetric relation, then not only is it the case that $R(A, B)$ is true just in case $R(B, A)$ is, but also it is the case that $\neg R(A, B)$ is true just in case $\neg R(B, A)$ is.

Obviously, more than one minimal situation can be constructed for any conditional statement. For example, to check $P \Rightarrow Q$, we can construct a situation containing P and then check for Q, or we can construct a situation containing $\neg Q$ and check for $\neg P$. However, the logical soundness of library structures guarantees that capture verification need only check one of these.

The other reason that capture verification need only check one minimal situation has to do with checking general statements. If general statements were checked in a brute force way, capture verification would be impractical for them. There are a large number (often an infinite number) of minimal situations that can be created to check a general statement, each differing only in the specific individuals mentioned. The well known law of generalization of constants for first order predicate calculus allows capture verification to check one situation containing anonymous individuals and infer that any situation differing only in the specific individuals mentioned will be the same.

The law of generalization [Bell & Machover 77, p.65] states that if it is possible to prove a theorem of the form $\alpha(x/c)$[3] from a set of formulas Φ and the constant c does not occur in Φ or α, then $\Phi \models \forall x[\alpha]$. Introducing new anonymous instances is equivalent to introducing previously unmentioned constants, thus any conclusion

[3]The notion $\alpha(x/c)$ means c is substituted for every occurrence of x in α.

that capture verification draws from a situation created with anonymous individuals will be true for all situations differing only in the specific individuals mentioned.

Monotonicity also has important consequences for the design process:

1. If a representation captures a set of statements, then any specialization will capture those statements.

2. If a representation captures a set of statements, then it captures every subset as well.

The combination of these properties means that any subset of the statements in a problem can be tested by capture verification at any time and if they are captured in the representation, subsequent representation design activity will not change this. Therefore, once capture verification indicates that a statement is captured it can be removed from the consideration of subsequent representation design.

If monotonicity is relaxed, this ceases to be the case and statements must be continually rechecked as representation design proceeds. Furthermore, there would no longer be any guarantee that the representation design process moved monotonically towards more fully constrained representations since redefining the representation of a concept could cause captured constraints to become uncaptured.

4.3.4 Interactions Between Knowledge Acquisition and Capture Verification

In our example problem, classifying *married* results in statements of its properties being added to the problem. Then, when classification of *married* terminates, capture verification removes these statements. One might think that these two processes are self defeating, i.e., one might wonder why these constraints are added simply to be immediately removed by capture verification. The reason is that concept introduction (the subject of Chapter 5) constructs equivalent alternative problem formulations. Statements captured in one formulation are not necessarily captured in another. Even though a statement acquired to specialize one representation will be captured by that representation, the statement may not be captured in an alternative

representation. Such statements are included to ensure that alternative formulations will be equivalent.

It is the function of knowledge acquisition to attempt to ensure that all relevant constraints appear in the problem, while the function of capture verification is to ensure that constraints are expressed in the problem statement or captured in the representation but not both.

The combination of these processes designs representations correctly for each of the following:

1. Cases where constraints that classification looks for are left out of a problem.

2. Cases where classification finds the constraints it is looking for in the problem.

3. Cases where constraints that classification looks for are in the problem but it does not recognize them.

The first case is illustrated by our example problem. The system acquires missing constraints, designs a representation to capture them, and then verifies that those constraints are indeed captured.

The second case would occur, for instance, if our example problem contained the statement

$$\forall x \forall y [married(x, y) \Rightarrow married(y, x)].$$

In this case, the system would not ask the user about the symmetry of *married* and this statement would be removed by capture verification after the classification of *married*.

The third case occurs because the techniques that classification uses to find a statement expressing a constraint are necessarily incomplete. In this case, knowledge acquisition will add redundant information to the problem. Capture verification, while also incomplete, is more powerful than the technique that classification uses, often identifying captured constraints that classification misses.

4.4 Summary

The representation design system captures constraints on a concept by representing it in terms of a library structure enforcing those constraints. The best structure (or set of structures) for representing a concept captures the most constraints on the concept without capturing constraints that are not true of the concept. Given a fixed vocabulary (set of concepts and sorts), classification identifies the best available library structures for representing each concept (and sort) in that vocabulary.

For classification, the library structures are organized into two hierarchies: one containing structures for representing concepts (Figure 4.1) and the other containing structures for representing sorts (Figure 4.3). Structures in the concept hierarchy capture constraints on concepts, for example, `function` captures single valuedness (i.e., $x = y \Rightarrow f(x) = f(y)$). Structures in the sort hierarchy capture constraints between or common to the members of a sort, for example, `fixed-size-set` is used to represent a sort whose members are sets and captures the constraint that all the sets are the same size.

Classification does knowledge acquisition, assuming that if the constraints it is looking for are not found in the problem statement, then they are worth asking the user about. When the user says that a concept has some property, the system adds a statement to the problem expressing that fact. Thus, classification often results in an expansion of the problem statement. The example given in this chapter is the classification of *married* in the small FAMILIES problem. The result is shown in Figure 4.7. The statements added during the classification of *married* are enclosed in the box.

Classifying *married* also results in a the following specialized representation for it:

`married: antitrans-sym-irr-rel(family-member,family-member).`

Capture verification is a constructive proof technique which uses a representation directly to determine if a constraint is captured in it. The technique relies on the following fact: A constraint is captured in a representation if and only if it is true in every situation that can be created with that representation. The technique is used to identify the statements that a classification effort has captured. The system removes captured statements from its problem description so that subsequent

representation design need not consider them. In our example, capture verification identifies the statements added by classification, and the system removes them. Thus, the statements in the box in Figure 4.7 are removed.

Chapter 5

Concept Introduction

Concept introduction extends the classification process by adding new concepts to a problem. The system introduces new concepts to *explore* the classification of *alternative problem formulations.* A new concept is introduced by defining it in terms of existing concepts in such a way that the semantics of a problem are not changed. When a new concept is introduced, a new representation is also introduced, giving the system access to different parts of the structure library, i.e., the new representation captures different constraints and has different specializations.

Gaining access to a different part of the structure library provides the opportunity to capture more problem constraints or to capture the same constraints more efficiently. Consider, for example, the introduction of *parents* during the design of the representation of the small FAMILIES problem: *Parents* is defined in terms of *child* as

$$\forall x \forall y [x \in parents(y) \Leftrightarrow child(x, y)].$$

This definition is equivalent to the more familiar

$$parents = \lambda y.\{x \mid child(x, y)\}.$$

However, as will be explained, the system requires new concepts definitions to be directly usable in reformulation. The first formula above has this property, while the second does not.

Introducing *parents* allows the system to view the concept "child" as a function into sets instead of as a relation, i.e., the new symbol *parents* is a function from family

members to sets of their parents. When *parents* is introduced, a representation is introduced for it and for its range: **parents** is defined in terms of **function** and **parent-set** is defined as a subtype of **set(family-member)**.

After introducing a new concept, the system creates a reformulation of the problem in terms of it. Alternative formulations are constructed by using the logical definition of a new concept to rewrite existing statements. For example, the definition of *parents* is treated as a rewrite rule to construct a formulation of the problem in terms of *parents* by rewriting every occurrence of *child* to *parents*. For instance, one statement that is rewritten in the small FAMILIES problem is

$$\forall x \; child(P, x) \Rightarrow x = R.$$

The formulation in terms of *parents* contains the equivalent statement

$$\forall x \; P \in parents(x) \Leftrightarrow x = R.$$

One fact in the formulation of small FAMILIES in terms of *child* is that every individual is a child of exactly two other individuals. In the formulation in terms of *parents* this fact becomes, "all sets of parents have exactly two members which allows classification to specialize **parent-set** in terms of **fixed-size-set**.

The **parents** representation, along with the specialized version of **parent-set**, captures the size constraint, a constraint that the system is not able to capture by specializing **child** because no specialization of **relation** captures size constraints. However, **parents** alone leaves some of the constraints uncaptured that **child** does capture.

Alternative formulations of a problem are compared based on how efficiently the problem constraints can be captured in them. The cost of a formulation is the cost of the concepts in it. The cost of a concept is the cost of the machinery capturing the constraints on that concept.

Consider an example of comparing two formulations. After introducing *parents*, the system compares the formulation in terms of it to the formulation in terms of *child*. It turns out that the cost of the machinery capturing the constraints on *parents* is less than the cost of the machinery for *child*. Since the statements mentioning other concepts in these two formulations are the same, the cost of the other concepts is

the same. Therefore, the cost of the formulation in terms of *parents* is less than the formulation in terms of *married* and the former formulation is preferred.

There are times when the system introduces alternative concepts and then decides that representations should be included for more than one alternative, i.e., the preferred formulation contains more than one alternative concept. For example, during the design of a representation for FAMILIES, an alternative for *parents* is introduced which is a function mapping a family member to his/her set of children. Let us call this function *children*. After a comparison analysis, it is decided that the problem representation should contain `parents` and `children` (but not `child`). From a sufficiency point of view only one of these concepts is necessary, however, it turns out that the representation containing both captures the problem's constraints more efficiently than a representation of either one alone. Thus, the system has decided that for this problem it is best to use multiple representations of the "child" concept.

There are also times when the system introduces alternative concepts and is forced to keep more than one equivalent concept. This happens when it is not able to reformulate a problem entirely in terms of an introduced concept.

Concepts are also introduced to reformulate mixed statements that are restrictions to specific statements about sets. The motivation for this has to do with the desire to design representations for problem classes. If a problem contains a statement restricting the number of individuals that can stand in some relation to a specific individual, we would like the system to design a representation that is not dependent on the number of individuals in the given statement. This way the representation can be used for another problem that has a similar constraint, but differs in the number of individuals involved. For example, we want the representation designed for a problem containing the statement

$$\forall x[child(P, x) \Leftrightarrow x = R]$$

to also work for a similar problem containing

$$\forall x[child(P, x) \Leftrightarrow x = R \vee x = S],$$

and so on.

The system accomplishes this type of generalization by reformulating restrictions to create statements about sets and then generalizing those statements to involve

arbitrary sets. For instance, to accomplish this in the above example, the system introduces the *children* function and reformulates. The reformulated versions of the two statements above are

$\exists y_1 \exists z_1 [children(y_1) = \{z_1\}]$
$\exists y_2 \exists z_2 \exists w_2 [children(y_2) = \{z_2, w_2\}]$

which the system generalizes to

$\exists y \exists z \; children(y) = z,$

where z is taken to be a set. Hence, a representation is designed to allow statements restricting the set of an individual's children to a constant set of any size.

Also, to ensure that alternatives for formulations containing mixed constraints are always preferred, the system assigns an infinite cost to mixed statements.

Classification extended by introduction is called *extended classification.* Extended classification is interesting because, while the two processes involved are fairly simple, the behavior of the combination can result in multiple reformulations of a problem. Several examples of this will be given later, including the sequence of introductions that results from the classification of *married*:

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. This function captures the following facts: not all individuals are married, each married couple is disjoint from all other married couples, married couples contain exactly two members.

5.1 Introduction Rules

Introduction is implemented as a collection of condition-action rules that are associated with nodes in the structure library hierarchies. A rule's condition part checks properties of the concept being classified. When the conditions are met, the action part introduces a new concept and its representation.

Figure 5.1 gives an example of an introduction rule which is associated with the node in Figure 5.2 for irreflexive, symmetric, antitransitive relations. When this node is reached while classifying a relation, the system reformulates that relation as a function into sets. Given a relation R, the rule introduces a function F_R into sets of the form $\{x \mid R(x,y)\}$. This is done to explore the use of the `function` and `set` representations (and their specializations) to design a more specialized representation than was possible for R.

> "For the relation $R : s_1 \times s_2$ introduce the new concept F_R defined as
> $\forall x \forall y [y \in F_R(x) \Leftrightarrow R(x,y)]$.
> Also introduce a representation for the range of F_R as
> `(deftype FR-ran specializes set(s2))`
> and introduce the representation F_R as
> `FR: function(s,FR-ran).`"

Figure 5.1: An example Introduction rule

In general, introduction rules specify new biconditionals (usually one, occasionally two) that define a new concept. By convention, we will always write such a definition with the new concept on the left. Introduction rules define representations for the concepts they introduce and often also introduce sorts along with their representations.

The system uses the logical definition of the new concept to create a new rewrite rule each time the introduction rule is applied. The example rule rewrites occurrences of the term $R(x,y)$ as $y \in F_R(x)$, where R is whatever relation is being classified when the introduction rule is applied. These rewrite rules create alternative problem formulations.

The form of logical definitions of new concepts is restricted to allow them to be used directly in reformulation. Such a definition must be expressed as a biconditional

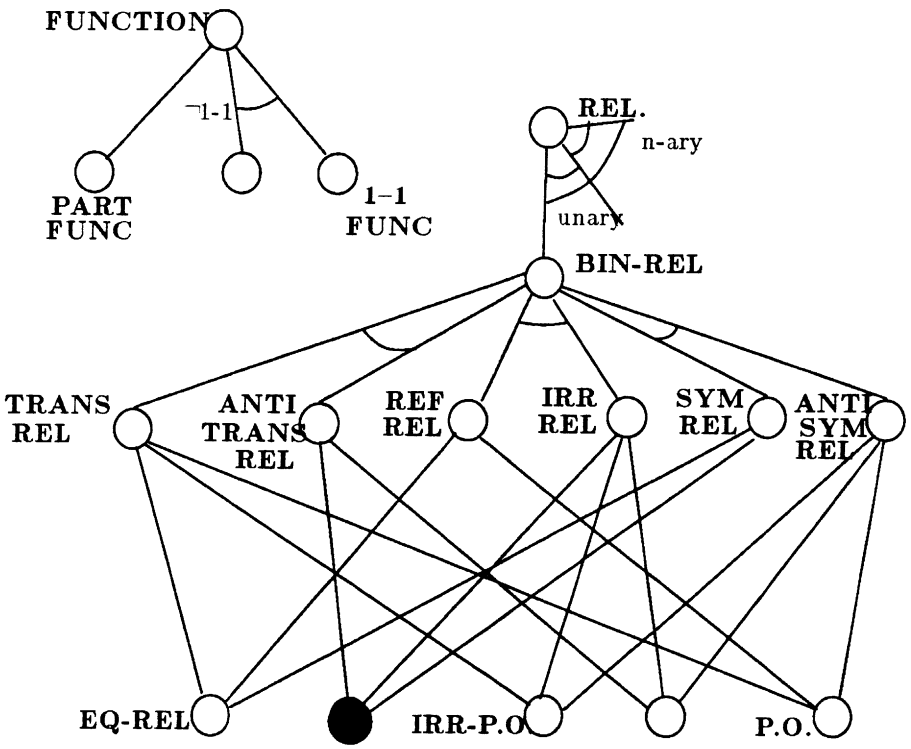


Figure 5.2: Node for irreflexive, symmetric, antitransitive relations.

between conjunctions of one or more atomic formulas. When both sides of a definition contain only one atomic formula, it is used as a rewrite rule as already described. When the left hand side is a conjunction, it is treated as a rewrite rule that replaces the atomic formula on the right by the conjunction of atomic formulas. For example, the definition

$$\forall x \forall y [couple\text{-}of(x) = couple\text{-}of(y) \wedge x \neq y \Leftrightarrow married(x, y)]$$

is treated as a rewrite rule that replaces atomic formulas of the form $married(x, y)$ by the conjunction on the left hand side. When this rule is used to rewrite the statement $married(N, P)$, the result is

$$couple\text{-}of(N) = couple\text{-}of(P) \wedge N \neq P.$$

When the right hand side of a definition is a conjunction, it is treated as a conditional rewrite. For example, consider the following definition of the function

non-empty-brothers:

$$\forall x \forall y[x = \textit{non-empty-brothers}(y) \Leftrightarrow x = \textit{brothers}(y) \wedge \textit{brothers}(y) \neq \emptyset].$$

This is interpreted as follows:

> "If a statement S contains a term of the form $\textit{brothers}(y)$ and it can be shown from the structure of S that the term denotes a non-empty brother set, then replace the term by $\textit{non-empty-brothers}(y)$."

The intention behind the phrase, "can be shown from the structure of S" is that the system will only attempt to show a condition by a syntactic analysis of S. For example, the above rewrite rule would be applied to the statement, $A \in \textit{brothers}(B)$, (where A and B are constants) because the statement unconditionally asserts that B's brother set is non-empty.

5.1.1 Capturing Constraints Between Sorts

Once the system introduces a sort, it attempts to capture taxonomic constraints between that sort and other sorts. This allows the system to capture some constraints between concepts, a type of constraint that classification can not capture.

As a simple example, suppose the system has introduced the sort of the elements for which a relation P is true and for the sort of the elements for which a relation Q is true. Let us call the representations of these sorts `P-dom` and `Q-dom` respectively. Then it will try to capture problem statements expressing a relationship between P and Q as a subsumption constraint between the types `P-dom` and `Q-dom`. For example, it captures the statement

$$\forall x[P(x) \Rightarrow Q(x)]$$

by making `P-dom` a subtype of `Q-dom`:

```
(deftype P-dom specializes Q-dom).
```

Through the process of introduction, the system can transform statements into a form recognized as a subsumption constraint and capture it in the type hierarchy of representations. For example, the constraint in the FAMILIES problem that the parents of an individual are married is initially expressed as

$$\forall x \forall y \forall z[\textit{child}(x,z) \wedge \textit{child}(y,z) \wedge x \neq y \Rightarrow \textit{married}(x,y)].$$

During design the system introduces two concepts: *parents* and *couple*. The latter is a mapping from an individual to the couple he/she is a member of. It also introduces representations for the ranges of these functions: *parent-set* and *married-couple* respectively. As will be described later in this chapter, because of properties that the system discovers of *parent-set*, it introduces *parent-set-of*, a function mapping an individual to the *parent-set* he/she is a member of. As a result, the above statement is rewritten as

$$\forall x \forall y [parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)],$$

which is recognized as expressing a subsumption relationship between *parent-set* and *couple*.[1] Therefore, it removes this statement from the problem and adds the following to the representation:

```
(deftype parent-set specializes couple).
```

Thus, through the process of introduction, the system has turned this statement, originally a constraint between *child* and *married*, into a subsumption constraint between *parent-set* and *married-couple* and captured the constraint efficiently in the type taxonomy of representations.

5.2 Soundness of Introduction

As we will see in the remainder of this chapter, many new concepts can be introduced during representation design. We would like to be sure that when a solution is found in the new formulation of a problem, it is a solution to the original problem. In other words, we would like to show that the introduction process is sound.

From a model theoretic point of view, a new concept is introduced by two actions: adding a new constant to the language of the problem and adding a new statement to the problem. It is well known that there can be no more models of a set of statements than of any of its subsets. Therefore, since concept introduction adds a new statement, the new problem can not have more models than the original. To

[1]This statement actually contains more information than the subtype constraint. In general, such statements can not be removed. However, in this case, the additional information turns out to be captured elsewhere in the representation, allowing the system to remove the statement.

ensure soundness, we show that no models of the original problem are eliminated by concept introduction.

This is done by ensuring that every introduction has the property that every model of the original problem can be extended to a model of the new problem which satisfies the new statement. Since the new models are extensions of models of the original problem, they are themselves models of the original problem. Since every model can be shown to have such an extension, no model of the original problem has been lost. When every introduction meets this restriction, the process as a whole is sound because every introduction step is sound.

This restriction is checked (by hand) for each introduction rule by interpreting the statement that the rule introduces as an abstract procedure to perform on models of the original problem to get the extended models. For example, the rule in Figure 5.1 can be shown to meet the restriction by showing how to treat the statement

$$\forall x \forall y [y \in F_R(x) \Leftrightarrow R(x, y)]$$

as a procedure that extends any model of the original problem.

Take any model of the original problem and add F_R to the set of function symbols of the model. Next, for each element, x in the domain of F_R, create a pair of the form $< x, \{y \mid R(x, y)\} >$. Add the union of these pairs to the domain of the model and designate this new set by the symbol F_R. The new model is an extension of the original model.

Note that when an introduction rule is conditional, we can assume its conditions while checking the restriction. Then we show that any model of the original problem that satisfies the conditions of the rule can be extended to a model of the new statement.

Each introduction rule in the system has been checked and shown sound in this fashion; it follows that the introduction process, as a whole, is sound.

5.3 Extended Classification

In order to present the intuition behind extended classification, we first explain the process under the assumption that no knowledge acquisition is required to design

representations. How extended classification is actually done for analytical reasoning problems is more complicated because they are usually missing information. This more complicated case is discussed below.

Concept introduction rules are attached to nodes in the library hierarchies. Extended classification collects the introduction rules it encounters while classifying a concept. When classification of the concept terminates, the system runs capture verification. Statements that can be captured by classification are general statements that mention only the concept being classified. If any statements of this form remain uncaptured after classification of the concept, all the introduction rules found while classifying it are checked.

All the rules whose conditions are satisfied are applied and each applied rule introduces a new concept. Note that the representation of a new concept may not capture statements that the original concept captures. Therefore, the formulations associated with new concepts contain reformulated versions of all the statements mentioning the original concept whether or not they are captured by the representation of the original concept. Extended classification proceeds by classifying each of the new concepts. When all the classification efforts terminate and capture verification is run in each alternative formulation, the alternatives are compared. Note that classification of an alternative can cause additional concepts to be introduced.

Consider as an example part of the extended classification of *married*. The introduction rule in Figure 5.1 is associated with the node for irreflexive, symmetric, antitransitive relations (shown shaded in Figure 5.3).

When applied, the rule introduces the concept *spouses*[2], a function from family members to their sets of spouses. *Spouses* is defined with the following statement:

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

A formulation of the problem in terms of *spouses* is generated when the concept is introduced. This formulation contains a new statement for every statement in the original formulation that mentions *married*, including those statements already captured by `married` (e.g., symmetry). Each new statement is logically equivalent to

[2]Recall that the system gives this concept a gensymed name. We have changed it to *spouses* for clarity of presentation. This is the case for the examples given throughout this chapter.

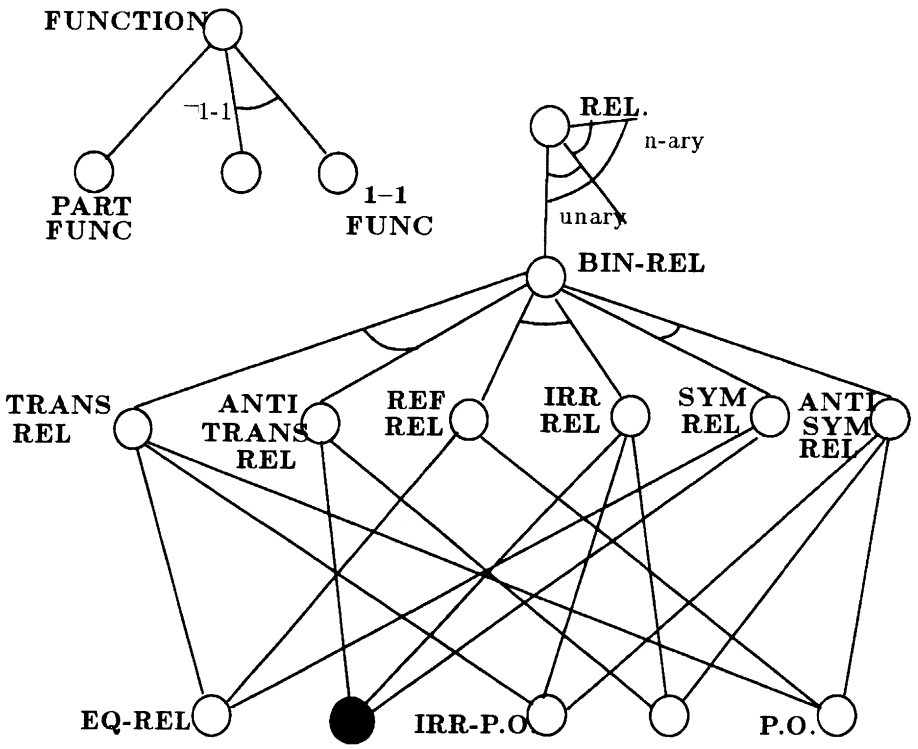


Figure 5.3: The node reached in classifying *married*.

its counterpart in the original formulation but is expressed in terms of *spouses*. For example, the statement,

$$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)],$$

has an equivalent counterpart in the new formulation:

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow x \in spouses(y)].$$

After *spouses* is introduced, extended classification proceeds by classifying it (it is the only alternative in this example). It turns out that as *spouses* is being classified, an alternative is introduced for it: *spouse*, a function from family members to their spouses. The system determines that *spouse* is preferable to both *spouses* and *married* (or their combination). Therefore, the formulation in terms of *spouse* alone is preferred.

5.3.1 Extended Classification in Incomplete Problems

Analytical reasoning problems often omit information necessary to solve them. Therefore, when classification of a concept terminates, the system can not assume that the introduction rules collected need only be tried when statements are left uncaptured. Instead, the system uses concept introduction not only as explained so far but also to extend the knowledge acquisition done by classification. For example, in a problem containing *married* which does not state that it is symmetric, classifying the concept acquires this property because there is a specialization of `relation` for symmetry. However, classifying *married* will not acquire the constraint that at most two people are married to each other because no specialization of `relation` captures a constraint like that.

The system always assumes that a problem is incomplete and applies introduction rules whether or not a specialized representation captures all the stated constraints on a concept because classifying a concept so introduced may uncover additional missing information. Therefore, whenever a classification effort terminates, if any introduction rules have been collected, *they are tried without regard to the problem statement.*

Classification of a concept introduced in this manner often yields constraints on the new concept. These are also constraints on the original concept. The statements of these new constraints are reformulated in terms of the original concepts because the system compares alternative concepts based on the cost of the constraints on them. For example, the FAMILIES problem is stated in terms of *child* and is missing the constraint on the number of parents of a family member. Classifying *child* does not uncover this missing constraint. However, a rule found while classifying *child* introduces the concept *parents* and classifying it does uncover this constraint because the range elements of *parents* are sets and `set` has a specialization that captures size constraints. When *parents* is classified, the following statement is added to the formulation of the problem in terms of *parents*:

$$\forall x \forall y \forall z [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \Rightarrow parents(x) = \{y, z\}].$$

Versions of a statement added in this way must be translated into the alternate formulations being investigated. This is done by returning to the logical definition of

the concept that is being classified and using this definition to derive a rewrite rule "that goes in the other direction." For example, when the statement above is added to the problem, the logical definition of *parents* in terms of *child*,

$$\forall x \forall y [x \in parents(y) \Leftrightarrow child(x, y)],$$

is used to derive a rule that rewrites the newly acquired statement to an equivalent statement in terms of *child*. The result is

$$\forall x \forall y \forall z [child(y, x) \wedge child(z, x) \wedge y \neq z \wedge child(w, x) \Rightarrow w = y \vee w = z].$$

5.3.2 Comparing Alternative Formulations

Decisions to choose from amongst alternative formulations require a comparison of their relative costs. The cost of a formulation is the sum of the costs of the concepts in it. The cost of a concept is the cost of the machinery capturing the constraints on it.

In general, some of a concept's constraints get captured by its representation designed by classification and the rest get captured by procedures operationalization generates for them. Thus, the cost of a concept is the cost of its representation designed by classification plus the cost of the machinery generated by operationalization.

To facilitate estimating the cost of a concept's representation, each library structure has a cost estimate associated with it. This makes it straightforward to estimate the cost of a concept's representation: the cost estimates of its component structures are simply added together. The cost estimate for all library types except `trans-rel` and `antitrans-rel` is 1; the estimate for both of these is n. These estimates are explained below.

The system has an evaluation function that, given a statement in implication normal form, estimates the cost of the procedures that operationalization will generate to capture that statement. A concept's estimate is related to the complexity of the procedures enforcing the constraints on that concept. The estimator takes advantage of the fact that its results will be used solely for the comparison of alternative formulations to simplify the estimation task. The estimates it produces need not be and are usually not accurate complexity measures.

The estimator also takes advantage of the restricted structure of the procedures generated by operationalization. These are always in the form of nested loops. For example, consider the following statement of transitivity:

$$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow R(x,z)].$$

The procedure that operationalization generates for this statement runs every time a new $< x, y >$ pair is added to a problem situation. When it runs, x and y are bound to constants, but z is unbound. Therefore, the procedure must search through the list of existing pairs related by R for any of the form $< y, z >$ and, for each such pair, adds a pair of the form $< x, z >$. This procedure performs one loop over the pairs related by R to enforce the transitivity constraint each time it is called.

Complexity estimates are computed for statements in as just illustrated: The variables in the first atomic formula are assumed to be bound. Then the procedure steps through the atomic formulas in the statement. For each, it checks to see if it contains unbound variables; then it assumes they are bound and continues. The number of loops is the number of atomic formulas with unbound variables in them. We will estimate the complexity of a procedure with no loops in it as 1, a procedure with one loop as n, a procedure with a loop nested inside another as n^2, and so on. Since, the procedure that operationalization will generate to capture transitivity has one loop in it, its complexity estimate is n.

The estimates for library structures are computed (by hand) in the same way: For each procedure, we count the nesting of loops to arrive at an estimate for that procedure. Then these estimates are added to obtain an estimate for the structure.

The estimation procedure is more complicated than simply counting the atomic formulas that have unbound variables in them for two reasons. First, an atomic formula with unbound variables does not always require a loop. For example, suppose the range of the function *spouse* is family members. Then the atomic formula $y = spouse(z)$ does not require a loop when y is unbound as long as z is bound. Therefore, the procedure is refined to check both the unbound variables and the representation of the concept mentioned in an atomic formula. Checking an atomic formula can yield one of three results: the formula can require no loops, one loop, or there can be no way of determining the possible bindings for its variables. When

the third case occurs in a statement, its cost is estimated as infinite. We have seen examples of the first two cases. As an example of the third case, consider estimating the cost of the following statement

$$\forall x \forall y \forall z [x = spouse(y) \wedge y = spouse(z) \Rightarrow x \neq spouse(z)].$$

Assuming that x and y are bound, note that z is unbound in the second atomic formula. In general, there is no way to determine all the individuals whose image under a function is equal to a particular individual. Therefore, there is no way to determine possible bindings for z in this statement and its cost is estimated to be infinite.

The second complication in the estimation procedure is that sometimes reordering the conjuncts in the antecedent makes the estimate lower. For example, the cost estimate of the statement above becomes 1 when its conjuncts are reordered because y is bound when we reach the second conjunct in

$$\forall x \forall y \forall z [y = spouse(z) \wedge x = spouse(y) \Rightarrow x \neq spouse(z)].$$

In light of this complication, the cost estimator considers all permutations of the antecedent atomic formulas and returns the lowest cost estimate.

The estimate for the complexity of a collection of statements is a polynomial. The worst this polynomial can be is the sum of the estimates for the individual statements. It can be less than the sum of the individual statements because sometimes constraints get captured as a side effect of capturing others. In light of this, an estimate is calculated for the uncaptured statements mentioning a concept by systematically operationalizing the statements starting with those having the lowest estimate and working up until all the statements are captured. The estimate for the collection is then the sum of the estimates for the statements actually operationalized. This estimate is added to the estimate for the concept's representation to obtain the total estimate for a concept.

Consider an example of comparing the cost of alternative formulations: A problem formulation in terms of *married* is shown in Figure 5.4 and an equivalent formulation in terms of *spouse* is shown in Figure 5.5.

Married is classified as an irreflexive, symmetric, antitransitive relation. The resultant representation, **married**, captures all but the size constraint. The cost estimate

Irreflexive:	$\forall x \neg married(x, x)$
Symmetric:	$\forall x \forall y [married(x, y) \Leftrightarrow married(y, x)]$
Antitransitive:	$\forall x \forall y \forall z [married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$
Size constraint:	$\forall x \forall y \forall z [married(x, y) \wedge married(x, z) \Rightarrow y = z]$

Figure 5.4: A set of statements expressing constraints on *married*.

Irreflexive:	$\forall x \; x \neq spouse(x)$
Symmetric:	$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)]$
Antitransitive:	$\forall x \forall y \forall z [x = spouse(y) \wedge z = spouse(y) \Rightarrow x \neq spouse(z)]$
Size constraint:	$\forall x \forall y \forall z [x = spouse(y) \wedge x = spouse(z) \Rightarrow y = z]$

Figure 5.5: Formulation of the statements in terms of *spouse*

for **married** is computed from the library as $n + 2$, i.e., the sum of the estimates for **irr-rel**, **sym-rel**, and **antitrans-rel**, which are 1, 1 and n respectively. The evaluator estimates the cost of the size constraint in Figure 5.4 as n, making the total cost estimate for *married* $2n + 2$.

Spouse is classified as a partial one-to-one function. **Spouse** captures the size constraint at a cost of 1.[3] An estimate of 1 is computed for antitransitivity in Figure 5.5 using the fact that *spouse* is represented in terms of **function** and reordering the conjuncts in the antecedent. The cost estimates calculated for other two uncaptured statements is also 1, making the total estimate for *spouse* 4. Therefore, *spouse* is preferred over *married*.

Decisions about which alternative formulations to keep are complicated by the fact that better problem formulations can often be found by including more than one alternative concept in a formulation. To allow the representation design system to find these, the comparison procedure is generalized to consider all subsets of a set of alternatives; it then chooses the subset of alternatives that has the lowest cost estimate.

When two subsets have the same cost estimate, the set whose concepts are closer to the initial concepts is preferred. When this rule does not establish a preference

[3]Recall that the representation design system determines which statements are captured by capture verification. For example, to determine that the size constraint is captured, capture verification creates a problem situation in which an anonymous individual **x** is the **spouse** of another individual **y** and **x** is the **spouse** of **z**. A procedure associated with **1-1 function** forces **y** to be equal to **z**. Thus, the consequent is true and the statement is captured.

between two subsets with the same cost estimate, one is chosen arbitrarily.

Normally the logical relationship between two alternative concepts is kept implicitly in the rewrite rules that generate one formulation from another. For example, the equivalence between *spouse* and *married* is kept in the rewrite rule that generates the formulation in terms of *spouse* from the formulation in terms of *married*. However, when a formulation contains alternative concepts, statements expressing the relationship between each alternative must be added to the formulation. For example, the system must add the statement,

$$\forall x \forall y [married(x, y) \Leftrightarrow y = spouse(x)],$$

to a formulation that includes both *married* and *spouse*.

The general procedure for computing the cost of a set of more than one alternative concept first computes the cost of each concept in isolation. To compute the cost of a set of concepts, it identifies those statements left uncaptured by all the concepts in the set. Next it determines the minimum cost of each of those statements, which requires comparing the costs of the alternative formulations of each statement. Finally it sums the costs of the representations of each concept, sums the minimum costs of the uncaptured statements, and adds to this the cost estimates for the statements relating the alternative concepts.

To illustrate this general comparison procedure, let us return to the example above and compare the cost of $\{married, spouse\}$ to $\{married\}$ and $\{spouse\}$. The cost of $\{married, spouse\}$ is computed as follows. All the statements in the example are captured by the combination of *married* and *spouse*. Thus, the cost so far is simply the sum of the costs of `married` and `spouse`, which is $n + 3$. Including the constraint between the two concepts increases the estimate to $n + 4$. Since the formulation in terms of *spouse* alone has cost 4, it is preferred.

5.3.3 An Example Choosing Multiple Alternatives

A case where it is more cost effective to keep multiple alternative concepts is illustrated by the extended classification of the *child* relation in a problem whose relevant statements are shown in Figure 5.6.

Irreflexivity:	$\forall x \neg child(x, x)$
Antisymmetry:	$\forall x \forall y [child(x, y) \Rightarrow \neg child(y, x)$
Antitransitive:	$\forall x \forall y \forall z [child(x, y) \wedge child(y, z) \Rightarrow \neg child(x, z)$
Size constraint:	$\forall x \forall y \forall z \forall w [child(y, x) \wedge child(z, x) \wedge y \neq z \wedge child(w, x) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$\forall x \neg child(A, x)$

Figure 5.6: Statements relevant to *child*.

Child is first classified; when this completes two introductions occur. One introduces *parents*; the other introduces *children*, a function from individuals to their set of children. After these are classified, it turns out that the size constraint in Figure 5.6 is captured by `parents` and the mixed constraint is captured by `children`. It also turns out that even though keeping both of these in the representation requires maintaining the relationship between them, this representation is still more cost effective than keeping any single concept, as shown below.

The classification effort begins with the *child* relation, which is classified as irreflexive, antisymmetric and antitransitive. Two introduction rules are found: one introduces *children*, the other introduces *parents*. *Children* is a function from family members to sets of the form $\{y \mid child(x, y)\}$. The range of *children* is a sort called *child-set*. Note that the rule that introduces this is the same rule that introduced *spouses* for *married*. The reason that two introductions were not tried for *married* is that the node for irreflexive, symmetric, antitransitive relations has only one introduction rule associated with it. This reflects the fact that there is no point in trying both rules when a relation is symmetric because the functions they create are equivalent.

The two introductions cause two new formulations of the problem to be generated. The formulation in terms of *children* is shown in Figure 5.7 and the formulation in terms of *parents* is shown in Figure 5.8.

Recall that the system contains a simplifier that partially canonicalizes statements. The new form of the mixed constraint in Figure 5.7 is the result of reformulation followed by simplification of the statement $\forall x \neg child(A, x)$. First, the reformulation rule introduced with *children* rewrites the statement as

$\forall x \; x \notin children(A)$.

This is simplified to

$children(A) = \emptyset$.[4]

Irreflexivity:	$\forall x\ x \notin children(x)$
Antisymmetry:	$\forall x \forall y [y \in children(x) \Rightarrow x \notin children(y)]$
Antitransitive:	$\forall x \forall y \forall z [y \in children(x) \wedge z \in children(y) \Rightarrow z \notin children(x)]$
Size constraint:	$\forall x \forall y \forall z \forall w [(x \in children(y) \wedge x \in children(z) \wedge y \neq z \wedge x \in children(w)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$children(A) = \emptyset$

Figure 5.7: Example problem rewritten in terms of *children*

Irreflexivity:	$\forall x\ x \notin parents(x)$
Antisymmetry:	$\forall x \forall y [x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z [x \in parents(y) \wedge y \in parents(z) \Rightarrow x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w [(y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$\forall x\ A \notin parents(x)$

Figure 5.8: Example problem rewritten in terms of *parents*

In comparing the concepts *child*, *children*, and *parents*, the system creates formulations for $\{child, parents\}$, $\{child, children\}$, $\{parents, children\}$, and $\{child, parents, children\}$. Initially, the formulations for the subsets with more than one concept contain only the constraints between the concepts in the subset. The formulation for $\{child, children\}$ contains the single statement:

$$\forall x \forall y [y \in children(x) \Leftrightarrow child(x, y)],$$

which is just the logical definition of *children*. The formulation for $\{child, parents\}$ contains the single statement:

$$\forall x \forall y [x \in parents(y) \Leftrightarrow child(x, y)].$$

The initial formulation for $\{children, parents\}$ contains the single statement:

$$\forall x \forall y [y \in children(x) \Leftrightarrow x \in parents(y)].$$

Initially, the formulation for $\{children, child, parents\}$ contains the three constraints,

[4]The rewriting system that implements this simplifier is discussed in more detail in appendix A.

$\forall x \forall y[x \in parents(y) \Leftrightarrow married(x,y)]$
$\forall x \forall y[y \in children(x) \Leftrightarrow married(x,y)]$
$\forall x \forall y[y \in children(x) \Leftrightarrow x \in parents(y)]$

The extended classification effort proceeds by classifying *parents* and *children*. The details of this are given later. Of interest for the current example are the four representations that result from the classification:

```
(deftype child-set specializes disjoint-set(family-member))
children: partial-function(family-member,child-set)
(deftype parent-set specializes
fixed-size-disjoint-set(2,family-member))
parents: function(family-member,parent-set)
```

To decide which formulation to keep, the system first estimates the cost of each concept. The `child` relation captures all but the size constraint and the mixed constraint at a cost of $n + 2$. The cost of the size constraint is n^2. Recall that we want the system to reformulate mixed statements as specific statements. To integrate this preference with the existing mechanism for exploring alternative formulations, the system estimates the cost of mixed statements as infinite. Therefore, the cost of $\{child\}$ is infinite.

The `parents` function (along with `parent-set`) captures the size constraint at a cost of 1 but does not capture any other constraints. Both the irreflexivity and antitransitivity constraints have estimates of 1. However, as in the formulation in terms of *child*, the cost of the mixed constraint in this formulation is estimated as infinite, making the total cost of *parents* infinite.

Initially, the `children` function (along with `child-set`) captures only what was the mixed constraint in the other formulations. The cost of these representations is 1. Irreflexivity and antisymmetry are estimated to have cost 1, the cost of antitransitivity is estimated as n, and the size constraint is estimated to have cost n^2. Therefore, the total cost estimate for $\{children\}$ is $n^2 + n + 3$.

The next step is for the system to compare the cost of all subsets of the three concepts that have finite estimates. There are three such subsets in this example: *children*, $\{children, parents\}$, and $\{child, children\}$.

The total cost of $\{children, parents\}$ is determined to be 6 as follows. The total cost of the representations is 2. The statements left uncaptured by both concepts are

irreflexivity, antisymmetry, and antitransitivity. Irreflexivity and antisymmetry each cost 1 in both formulations. Antitransitivity has a cost of 1 in the formulation in terms of *parents* and n in the formulation in terms of *children*. Therefore, the system includes the version expressed in terms of *parents* in the formulation containing both concepts. The cost of the constraint between *parents* and *children*, i.e.,

$$\forall x \forall y [x \in parents(y) \Leftrightarrow y \in children(x)],$$

is estimated to be 1.

Finally the costs of the three alternatives are compared: $\{children\}$ has cost n^2+n+3, $\{child, children\}$ has cost $n^2 + n + 4$ and $\{children, parents\}$ has cost 6. Thus, the $\{children, parents\}$ alternative is chosen.

Figure 5.9 shows the collection of statements associated with this alternative. The statements in this figure are those whose cost estimates were used to calculate the total cost. Note that since irreflexivity and symmetry have the same cost estimate in both formulations, the statements expressing those constraints are chosen arbitrarily from the *parents* formulations.

Irreflexivity:	$\forall x \; x \notin parents(x)$
Antisymmetry:	$\forall x \forall y [x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z [x \in parents(y) \wedge y \in parents(z) \Rightarrow x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w [(y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x)) \Rightarrow w = y \vee w = z]$
A mixed constraint:	$children(A) = \emptyset$
Interconcept constraint:	$\forall x \forall y [y \in children(x) \Leftrightarrow x \in parents(y)]$

Figure 5.9: Formulation in terms of both *parents* and *children*.

5.4 Special Types of Introduction Rules

In general, extended classification applies introduction rules as has been explained: All the rules associated with nodes reached in classification of a concept are collected. When classification terminates all rules are tried and the resulting alternatives are compared. This procedure is required because, in general, alternative concepts capture different subsets of the constraints on a concept.

Currently there are two types of special introduction rules for which the general comparison procedure can be circumvented. One is called a *replacement rule*. An introduction rule is a replacement rule in either of the following two cases:

1. The rule introduces an alternative concept whose representation captures the same constraints as the representation of the existing concept but is more efficient.[5]

2. The rule introduces an alternative concept whose representation is just as efficient and captures a superset of the constraints of the existing concept.

Because these rules replace one concept with another, when classification of a concept reaches a node with such a rule, classification of the concept need not continue. Therefore, replacement rules are applied immediately, in contrast to the normal process of waiting until classification of the existing concept terminates.

One example of a replacement rule is shown in Figure 5.10. It introduces the new function f' which is defined so that where the value of f is the singleton set $\{A\}$, the value of f' is A.

"When a sort S is represented as
`(deftype S specializes fixed-size-set(1,s1))`,
then introduce a function that "flattens" S as follows. Let f be the defining function for S and suppose f is represented as
`f: function(s2,S)`.
Then introduce f' with the definition
$\forall x \forall y [x = f'(y) \Leftrightarrow x \in f(y)]$,
represented as
`f': function(s2,s1)`."

Figure 5.10: Rule that introduces a function into individuals when an existing function is into sets of size 1.

This rule is attached to the `fixed-size-set` node in the sort hierarchy. An example of this rule's use is given in the next section during the classification of *spouse-set*. Since non-empty spouse sets all have size one, the rule introduces the function *spouse*, from an individual to that individual's spouse.

[5]Note that the efficiency difference may be more fine grained that the cost estimation procedure can measure.

The rule is a replacement rule because case one above applies: `f'` captures the same constraints as the combination of `f` and `S` and `f'` is more efficient, removing a "level of indirection" through the singleton sets.

The second special type of introduction rule is called an *extension rule.* An extension rule introduces a concept that is not an alternative to any existing concept but which results in a more efficient representation. An example of this type of rule is attached to the `disjoint-set` node in the sort hierarchy. When the members of a sort are found to be disjoint sets, this rule introduces a function from the members of those sets into those sets. For example, members of *parent-set* are found to be disjoint and so this rule introduces a function *parent-set-of*, mapping an individual into the parent set that he/she is a member of. For instance, if A and B are parents of C, then $\textit{parent-set-of}(A) = \textit{parent-set-of}(B) = \{A, B\}$.

The function introduced by this rule extends a problem representation so that it captures disjointness efficiently. A straightforward way to enforce disjointness is every time a new set of this type is created (or an existing set is modified), search a list of all other sets (of this type) to check for an intersection. If an intersection is found, combine the two sets involved. Introducing the new function takes advantage of the fact that every individual is in at most one of the sets of this type to avoid ever having to perform the above check to enforce disjointness.

Having direct access from an individual to the set that it is a member of increases the efficiency of the representation in other ways as well. For example, introducing *parent-set-of* in the FAMILIES problem reformulates the statement,

$$\forall x \forall y \forall c[\ x \in \textit{parents}(c) \wedge y \in \textit{parents}(c) \wedge x \neq y \Rightarrow \textit{couple}(x) = \textit{couple}(y) \wedge x \neq y]$$

as

$$\forall x \forall y \forall c[\ \textit{parent-set-of}(x) = \textit{parents}(c) \wedge \textit{parent-set-of}(y) = \textit{parents}(c) \wedge x \neq y \Rightarrow \textit{couple}(x) = \textit{couple}(y) \wedge x \neq y]. \quad (1)$$

From this statement, the simplifier first produces

$$\forall x \forall y[\ (\textit{parent-set-of}(x) = \textit{parent-set-of}(y) \wedge x \neq y) \Rightarrow (\textit{couple}(x) = \textit{couple}(y) \wedge x \neq y)].$$

The above statement is then further simplified to

$$\forall x \forall y [parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)].$$

As noted, this statement is recognized as a subsumption constraint and is captured in the type hierarchy.

5.5 Extended Classification of Married

This section illustrates the extended classification process with the classification of *married*. We begin with a summary of the main steps. After *married* is classified as an irreflexive, symmetric, antitransitive relation, the function *spouses* is introduced along with *spouse-set*, a sort for the range of *spouses*. During the classification of *spouses*, a partial function, called *non-empty-spouses*, is introduced. This maps from family members to non-empty spouse sets. Next, since range elements of *non-empty-spouses* all have size one, *non-empty-spouses* is replaced by the concept *spouse*, mapping a family member to his/her spouse. *Spouse* is classified as a partial one-to-one function and is determined to be its own inverse. This causes the partial function *couple* to be introduced, which maps an individual to the couple he/she is a member of.

5.5.1 Introduction of Spouses

As discussed in the last chapter, *married* is classified as a irreflexive, symmetric, antitransitive relation, placing us at the shaded node in Figure 5.3. As noted earlier, the system finds and applies the rule associated with this node which was shown in Figure 5.1. The result of applying this rule is to introduce *spouses* as

$$\forall x \forall y [y \in spouses(x) \Leftrightarrow married(x, y)].$$

Representations are defined for *spouses* and its range (*spouse-set*) as

```
(deftype spouse-set specializes set(family-member))
spouses: function(family-member,spouse-set).
```

The logical definition above is used to generate a formulation of the problem in terms of *spouses*. The original formulation of the problem is shown in Figure 5.11 and the formulation in terms of *spouses* is shown in Figure 5.12.

$married(Q, P)$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \neg married(x, x)$
$\forall x \forall y[married(x, y) \Leftrightarrow married(y, x)]$
$\forall x \forall y \forall z[married(x, y) \wedge married(y, z) \Rightarrow married(x, z)]$

Figure 5.11: Statements from small FAMILIES problem mentioning *married*

$P \in spouses(Q)$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow y \in spouses(x)]$
$\forall x \; x \notin spouses(x)$
$\forall x \forall y[y \in spouses(x) \Leftrightarrow x \in spouses(y)]$
$\forall x \forall y \forall z[x \in spouses(y) \wedge y \in spouses(z) \Rightarrow x \notin spouses(z)]$

Figure 5.12: Statements from small FAMILIES problem rewritten in terms of *spouses*

5.5.2 Classification of Spouses

Next, classification of *spouses* is initiated. This requires prior classification of its range *spouse-set* because the sorts a concept is defined over are always classified before the concept. Since *spouse-sets* are sets, classification begins at the **set** node in the sort hierarchy. The first question is whether *spouse-sets* all have a fixed size. The representation design system can not find any information about this in the problem statement, so it asks the following question:

> **Do sets of the form {y| married(x,y)} all have the same size?**
> No.

The next question is whether all members have the same maximum size. Again the system asks.

> **Do sets of the form {y|married(x,y)} all have the same maximum size?** Yes, 1.

Next is the question of whether their are empty spouse sets:

> **Can sets of the form {y| married(x,y)} be empty?** Yes.

At this point classification has reached the leaf node for empty members (Figure 5.13) which is checked for introduction rules. The following rule is found:

"When a sort S, whose members are sets, contains empty members, introduce a partial function into the non-empty members of S as follows. Let f be the defining function for S and suppose the representation for f is
`f: function(s1,S).`
Then define the function *non-empty-f* as
$\forall x \forall y[y = \textit{non-empty-}f(x) \Leftrightarrow y = f(x) \wedge f(x) \neq \emptyset]$
$\forall x \forall y[\textit{non-empty-}f(x) = \perp \Leftrightarrow y = f(x) \wedge f(x) = \emptyset]$.
Also introduce representations for *non-empty-f* and its range:
`(deftype non-empty-S specializes S)`
`non-empty-f: partial-function(s1,non-empty-S).`"

In this case, the rule introduces a sort whose members are non-empty spouse sets with the definition,

$$\forall x \forall y[x = \textit{non-empty-spouses}(y) \Leftrightarrow x = \textit{spouses}(y) \wedge \textit{spouses}(y) \neq \emptyset].$$

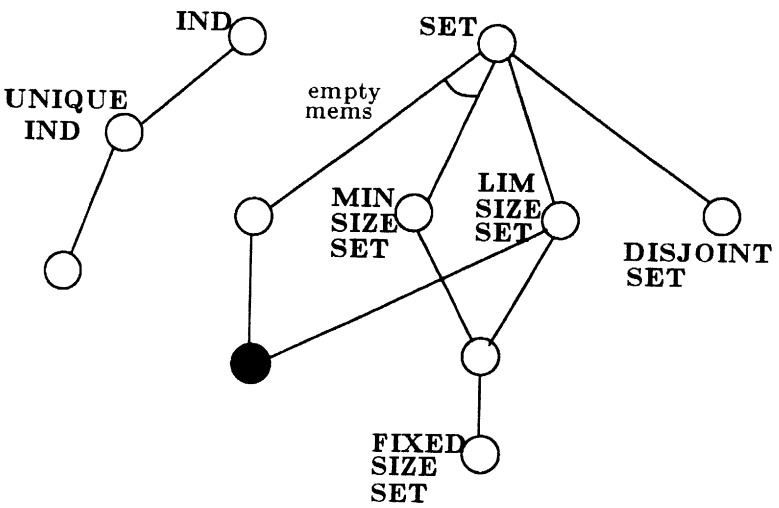


Figure 5.13: Node reached while classifying *spouse-set*

This causes the formulation of the problem shown in Figure 5.14 to be generated and classification of *non-empty-spouse-set* begins. The first question that classification raises is whether individuals of this sort all have the same size. The representation design system deduces that they all have size one from the following three facts:

1. *non-empty-spouse-set* is a subsort of *spouse-set*
2. members of the collection *spouse-set* have maximum size one
3. members of the collection *non-empty-spouse-set* are non-empty

$P \in non\text{-}empty\text{-}spouses(N)$
$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow y \in non\text{-}empty\text{-}spouses(x)]$
$\forall x\ x \notin non\text{-}empty\text{-}spouses(x)$
$\forall x \forall y[y \in non\text{-}empty\text{-}spouses(x) \Leftrightarrow x \in non\text{-}empty\text{-}spouses(y)]$
$\forall x \forall y \forall z[x \in non\text{-}empty\text{-}spouses(y) \wedge y \in non\text{-}empty\text{-}spouses(z) \Rightarrow x \notin non\text{-}empty\text{-}spouses(z)]$

Figure 5.14: The problem rewritten in terms of *non-empty-spouses*

5.5.3 Introduction of Spouse

This classification effort has now reached the node for fixed size sets (shown shaded in Figure 5.15). The replacement rule of Figure 5.10 is associated with this node. The rule applied in this case replaces the function *non-empty-spouses* with the function *spouse*. The rewritten version of the problem is shown in Figure 5.16.

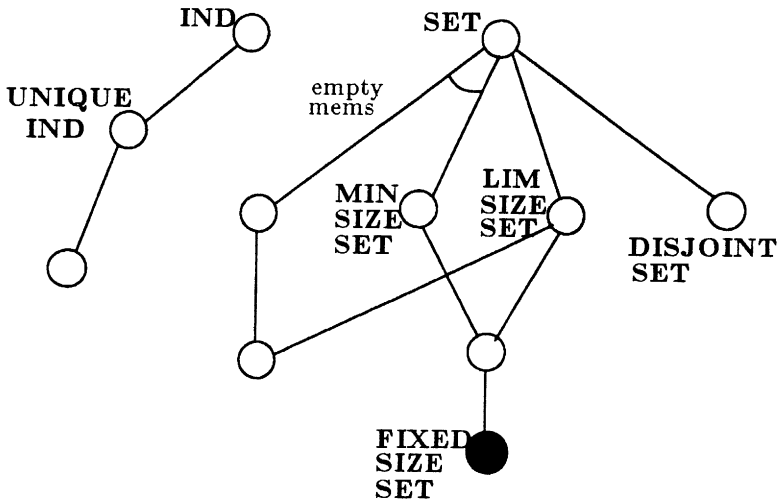


Figure 5.15: Sort hierarchy with node for fixed size set shaded.

$\forall x\ x \neq spouse(x)$
$\forall x \forall y[x = spouse(y) \Leftrightarrow y = spouse(x)]$
$\forall x \forall y \forall z[x = spouse(y) \wedge z = spouse(y) \Rightarrow x \neq spouse(z)]$
$\forall x \forall y \forall z[x = spouse(y) \wedge x = spouse(z) \Rightarrow y = z]$
$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow x = spouse(y)]$

Figure 5.16: Statements expressing constraints on *married* formulated in terms of *spouse*

Notice that this rule would not have been applied if members of the col-

lection *non-empty-spouse-set* had any other size but one. In that case, `non-empty-spouse-set` would be redefined in terms of `fixed-size-set`.

5.5.4 Classification of Spouse

The function *spouse* is classified as partial and one-to-one. The system determines that *spouse* is partial from the fact that *non-empty-spouses* is. It determines that *spouse* is one-to-one from the following problem statement, derived by the introduction process:

$\forall x \forall y [x = spouse(y) \Leftrightarrow y = spouse(x)]$.

It also determines from this statement that *spouse* is its own inverse.

This classification effort terminates at the nodes for partial function and one-to-one function (shaded in Figure 5.17).

5.5.5 Introduction of Couple

The node for one-to-one functions has the following introduction rule associated with it:

> "If a one-to-one function f is its own inverse, introduce a sort for sets of the form $\{x, f(x)\}$, and a function f' mapping an individual into its pair as follows.
> Let the representation of the function be
> `f: 1-1-function(s,s).`
> Introduce the function f' as
> $\forall x \forall y [f'(y) = f'(x) \wedge x \neq y \Rightarrow x = f(y)]$.
> Then introduce a representation `f'-pair` for the range of f' as
> `(deftype f'-pair specializes fixed-size-disjoint-set(2,s)).`
> Introduce the representation `f'` as
> `f': function(s,f'-pair).`"

This rule exploits the fact that a function being one-to-one and it being its own inverse means that the sets so introduced are disjoint from each other. In the case of our example, the rule introduces the concept of married couple and produces the formulation of the problem shown in Figure 5.18. Note that the statement of irreflexivity in this figure is equivalent to *true*. Therefore, it is removed.

The next step is the classification of *couple* and its range, call it `married-couple`.

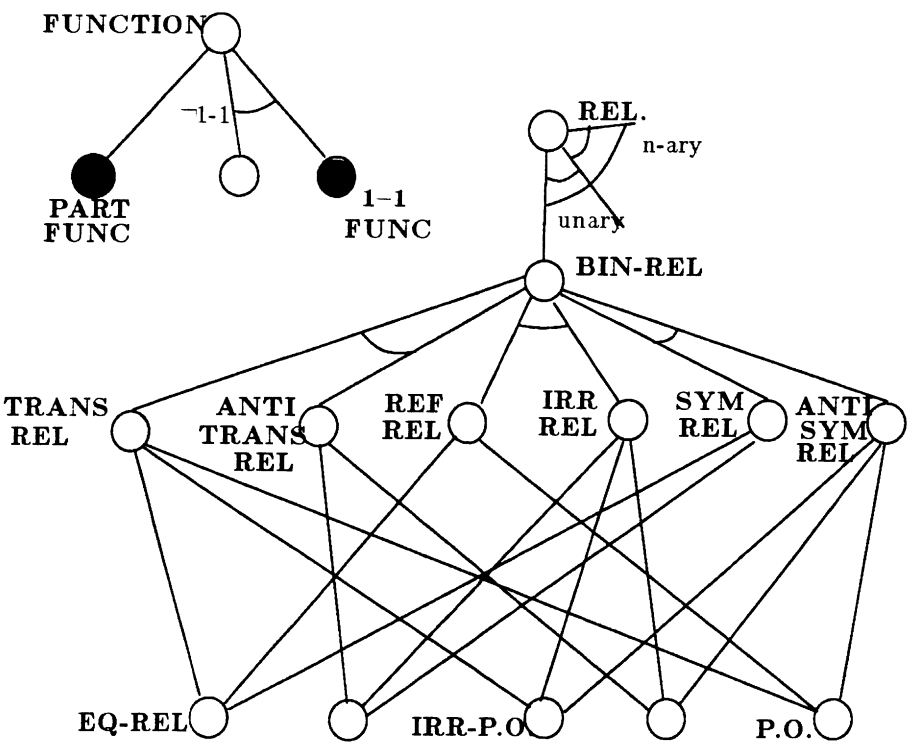


Figure 5.17: Nodes reached in classifying *spouse*.

$$couple(P) = couple(Q) \wedge P \neq Q$$
$$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge y \neq x]$$
$$\forall x[couple(x) \neq couple(x) \vee x = x]$$
$$\forall x \forall y[couple(x) = couple(y) \wedge y \neq x \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$$
$$\forall x \forall y \forall z[(couple(y) = couple(x) \wedge x \neq y \wedge couple(z) = couple(y) \wedge y \neq z) \Rightarrow couple(z) \neq couple(x) \vee x = z]$$

Figure 5.18: The problem rewritten in terms of *couple*

The classification of **married-couple** is trivial since the associated representation is already defined to have fixed sized disjoint members. The classification of *couple* yields the following definition

couple: partial-function(family-member,married-couple).

Couple is determined to be partial from the fact that *spouse* is.

The final step in this example is to compare the alternative formulations: when *couple* is compared with *spouse*, the system determines that *couple* is cheaper; *couple* is also found to be cheaper than *married*.

All of the statements in this formulation, except the one mentioning *child*, are captured by the combination of **couple** and **married-couple**.

One final note. The fact that *couple* is partial is not used in this problem. However, suppose the problem also contained the statement, "A is not married," i.e.,

$$\forall x \neg married(A,x).$$

When *spouses* is introduced, the following form of this statement is added to the problem,

$$spouses(A) = \emptyset.$$

Then when *non-empty-spouses* is introduced, the following statement is produced,

$$non\text{-}empty\text{-}spouses(A) = \perp,$$

which is eventually rewritten as

$$couple(A) = \perp.$$

When a problem situation is created from a statement like this one, the library type **partial-function** records that the domain element involved has no image under

this function. If a subsequent attempt is made to assign *non-empty-spouses*(A) a value, a contradiction is signalled.[6]

5.5.6 Summary of Extended Classification of Married

Recall that before the extended classification of *married*, the small FAMILIES problem was in the state shown in Figure 5.19. Also recall the following statements are added by knowledge acquisition during the classification of *married* (and before introducing *spouses*):

$P :, family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c[married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$
Query: find-all x: $parent(S, x)$

Figure 5.19: The small FAMILIES problem before classification of *married*.

$\forall x \neg married(x, x)$
$\forall x \forall y[married(x, y) \Leftrightarrow married(y, x)]$
$\forall x \forall y \forall z[married(x, y) \wedge married(y, z) \Rightarrow \neg married(x, z)]$.

For the sake of clarity in the example of the extended classification of *married*, we assumed that the these statements were already in the problem. Note that the size constraint is not present in Figure 5.19 and is not acquired during classification of *married*. After *spouses* is introduced, the classification of *spouse-set* uncovers the size constraint.

Figure 5.20 shows the state of the problem after extended classification of *married* and Figure 5.21 shows the representation. The overall effect of this has been to acquire missing constraints and to capture several statements with the specialized representation of *couple*. The captured statements are enclosed in a box in Figure 5.20.

[6]Or if *non-empty-spouses*(A) already has a value when this statement is added to a problem situation, a contradiction is signalled.

$P : family\text{-}member$, $Q, family\text{-}member$),
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$couple(P) = couple(Q) \wedge P \neq Q$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \exists c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge y \neq x]$
$\forall x \forall y \forall c[couple(y) = couple(x) \wedge y \neq x \wedge child(x, c) \Rightarrow child(y, c)]$

$\forall x \forall y[couple(x) = couple(y) \wedge y \neq x \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[(couple(y) = couple(x) \wedge x \neq y \wedge couple(z) = couple(y) \wedge y \neq z) \Rightarrow couple(z) \neq couple(x) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge couple(x) = couple(z) \Rightarrow y = z]$

Query: find-all $x \mid parent(S, x)$

Figure 5.20: The state of the small FAMILIES problem after extended classification of *married*.

```
(deftype family-member specializes unique-individual disjoint)
child:  relation(family-member,family-member)
parent:  relation(family-member,family-member)
(deftype married-couple specializes
fixed-size-disjoint-set(2,family-member))
couple:  partial-function(family-member,married-couple)
```

Figure 5.21: Representation of the small FAMILIES problem after extended classification of *married*.

5.6 Extended Classification of Child

This section takes up the extended classification of *parents* and *children*, concepts introduced during the classification of *child*. In addition to presenting another example of extended classification, this section illustrates that there can be interaction between the classification of closely related concepts. This can affect classification and knowledge acquisition. In this example, we will see that the classification of members of the sort *parent-set* as disjoint affects the classification of and knowledge acquisition for *child-set*, the sort of the range of *children*.

We begin with a summary: The *child* relation is classified as irreflexive, antisymmetric, and antitransitive. The node reached as a result of this has two introduction rules associated with it which introduce *parents* and *children*. In this example, we arbitrarily begin with the classification of *parents*. *Parent-set* (the range of *parents*) is classified as a sort whose elements are fixed size disjoint sets.

When the two new concepts are introduced, alternative formulations of the problem in terms of both of them are introduced. Also a formulation containing both concepts is introduced. This formulation contains a statement expressing the relationship between *parents* and *children*. The representation design system uses this relationship and the fact that *parent-sets* are disjoint to deduce that *child-sets* are also disjoint. Eventually the functions *parents'* and *children'* are introduced. *Parents'* is a one-to-one function from *child-set* to *parent-set*, while *children'* is a one-to-one function from *parent-set* to *child-set*. These two are found to be inverses of each other. These discoveries contribute to the preference for a formulation of the problem in terms of both *parents'* and *children'*.

The details of this example follow. This time we will suppress the knowledge acquisition done by the system and simply begin with the statements shown in Figure 5.22. These are the statements from the small FAMILIES and acquired by knowledge acquisition that mention *child*.

Irreflexivity:	$\forall x \neg child(x, x)$
Antisymmetry:	$\forall x \forall y [child(x, y) \Rightarrow \neg child(y, x)]$
Antitransitivity:	$\forall x \forall y \forall z [child(x, y) \wedge child(y, z) \Rightarrow \neg child(x, z)]$
Size constraint:	$\forall x \forall y \forall z \forall w [child(y, x) \wedge child(z, x) \wedge y \neq z \wedge child(w, x) \Rightarrow w = y \vee w = z]$
The mixed constraint:	$\forall x\ child(P, x) \Leftrightarrow x = R$
child/couple:	$\forall x \forall y \forall z [child(y, x) \wedge child(z, x) \wedge y \neq z \Rightarrow couple(y) = couple(z) \wedge y \neq z]$

Figure 5.22: Statements from small FAMILIES mentioning *child*, including those acquired during classification.

5.6.1 Classification of Parents

Parents is a function mapping family members to their set of parents. Figure 5.23 gives the problem statement rewritten in terms of *parents*. Before *parents* can be classified, *parent-set* must be, because the sorts that a concept is defined over must be classified before the concept. This begins at the top of the sort hierarchy. Members of the sort *parent-set* are sets, so the system asks whether parent sets have a fixed size (yes, everyone has two parents).

Irreflexivity:	$\forall x\ x \notin parents(x)$
Antisymmetry:	$\forall x \forall y [x \in parents(y) \Rightarrow y \notin parents(x)]$
Antitransitive:	$\forall x \forall y \forall z [x \in parents(y) \wedge y \in parents(z) \Rightarrow \neg x \notin parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x) \Rightarrow w = y \vee w = z]$
The mixed constraint:	$\forall x\ P \in parents(x) \Leftrightarrow x = R$
parents/couple:	$\forall x \forall y \forall z [y \in parents(x) \wedge z \in parents(x) \wedge y \neq z \Rightarrow couple(y) = couple(z) \wedge y \neq z]$

Figure 5.23: Example problem rewritten in terms of *parents*

Continuing, we come to the question of whether parent sets are disjoint (yes). This places classification at the nodes for fixed size sets and disjoint sets (shown shaded in Figure 5.25). Recall that there is an extension rule associated with the node for disjoint sets. The rule is shown in Figure 5.24.

In the case of our example, the rule introduces the function *parent-set-of* a mapping from a family member to the parent set he/she is a member of. Along with this

"When the range S of a function f has elements that are disjoint sets, introduce a function mapping an individual of the element sets in to the set it is a member of as follows. Let S be represented as

`(deftype S specializes disjoint-set(s1)).`

Introduce the function *map-S* as

$\forall x \forall y [map\text{-}S(x) = f(y) \Leftrightarrow x \in f(y)]$,

with representation

`map-S: function(s1,S).`"

Figure 5.24: Rule that, given a sort of disjoint sets, introduces a function mapping an individual to the unique disjoint set it is a member of.

introduction is a rule that rewrites terms of the form $x \in parents(y)$ to the form $parent\text{-}set\text{-}of(x) = parents(y)$. The result of rewriting the problem is shown in Figure 5.26.

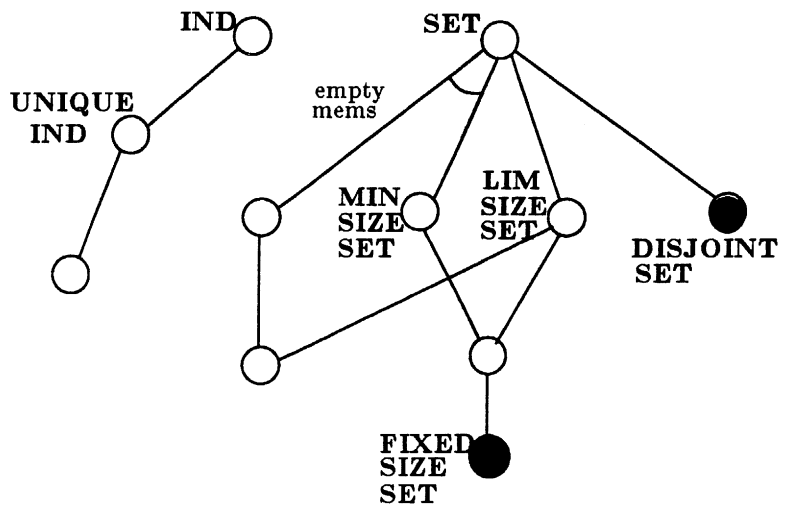


Figure 5.25: Node reached in classifying *parent-set*.

This rule also rewrites the constraint between *parents* and *children* in the formulation of the problem involving both:

$\forall x \forall y [x \in children(y) \Leftrightarrow y \in parents(x)]$;

it is rewritten as

$\forall x \forall y [x \in children(y) \Leftrightarrow parent\text{-}set\text{-}of(y) = parents(x)]$.

This completes the classification of *parent-set*, so the representation design system

Irreflexivity:	$\forall x\ parent\text{-}set\text{-}of(x) \neq parents(x)$
Antisymmetry:	$\forall x \forall y[parent\text{-}set\text{-}of(x) = parents(y) \Rightarrow parent\text{-}set\text{-}of(y) \neq parents(x)]$
Antitransitive:	$\forall x \forall y \forall z[parent\text{-}set\text{-}of(x) = parents(y) \wedge parent\text{-}set\text{-}of(y) = parents(z) \Rightarrow parent\text{-}set\text{-}of(x) \neq parents(z)]$
Size constraint:	$\forall x \forall y \forall z \forall w[parent\text{-}set\text{-}of(y) = parents(x) \wedge parent\text{-}set\text{-}of(z) = parents(x) \wedge y \neq z \wedge parent\text{-}set\text{-}of(w) = parents(x) \Rightarrow w = y \vee w = z]$
A mixed constr:	$\forall x\ parent\text{-}set\text{-}of(A) \neq parents(x)$
parents/couple:	$\forall y \forall z[parent\text{-}set\text{-}of(y) = parent\text{-}set\text{-}of(z) \wedge y \neq z \Rightarrow couple(y) = couple(z)]$

Figure 5.26: Example problem rewritten in terms of *parent-set*

returns to the classification of *parents* which is classified as a function that is not one-to-one.

5.6.2 Introduction of Children'

Classification of *parents* ends at the nodes for functions that are not one-to-one (shown shaded in Figure 5.27). This node has the following introduction rule associated with it:

> "If a function that is not 1–1 has range elements that are disjoint sets, then introduce a representation for sets of domain elements that map to the same range element. By definition, the sets in this new set are also disjoint, so introduce a 1–1 function from the range of the original function to the new set. More precisely, let f be a function that is not 1–1 and let it be represented as `f: function(s1,S)`
> and let S be represented as
> `(deftype S specializes disjoint-set(s2))`.
> Then introduce the function f' as
> $\forall x \forall y[y \in f'(x) \Leftrightarrow x = f(y)]$,
> introduce a representation for the range of f' as
> `(deftype f'-ran specializes disjoint-set(s1))`,
> and introduce a representation for f':
> `f': function(s2,f'-ran)`."

To see that the sets introduced by this rule are disjoint, consider the current case. The range elements of *parents* are disjoint sets. If we collect together sets of individuals that have the same parents, i.e., sets whose elements have the same image under *parents*, then we have created a set of disjoint sets. So the rule introduces the sort

children'-ran whose members are sets of family members with the same parents. It then introduces a 1-1 function that we call *children'* which maps parent sets to children sets such that if an individual x is a member of the parents of y, then the *children'-ran* of y equals the *children'* of the *parent-set-of*(x), i.e.,

$$\forall x \forall y[y \in children'(x) \Leftrightarrow x = parents(y)].$$

As usual, this definition is used as a rewrite rule that generates a new formulation of the problem. It also generates a new version of the statement,

$$\forall x \forall y[x \in children(y) \Leftrightarrow parent\text{-}set\text{-}of(y) = parents(x)].$$

(Recall that this is the result of rewriting the constraint between *parents* and *children*, done when the function *parent-set* was introduced.) The new version of this statement is

$$\forall x \forall y[x \in children(y) \Leftrightarrow x \in children'(parent\text{-}set\text{-}of(y))],$$

i.e.,

$$\forall y[children(y) = children'(parent\text{-}set\text{-}of(y))].$$

An important inference is made from this statement: the range of *children'* is determined to be the same as the range of *children* (i.e., the *child-set* = *children'-ran*). From this fact, the representation design system chooses one of the sorts to be the representative for both of them. Note that, because of this, the representation design system now knows that members of the sort *child-set* are disjoint.

5.6.3 Classification of Children'

The system now classifies *children'* which, as usual, requires prior classification of its range. This has just been determined to be *child-set*. Thus, the system begins with the classification of *child-set*. Members of this sort do not have a fixed size nor do

$$\forall x \forall y[\mathit{child\text{-}set\text{-}of}(x) = \mathit{children}(y) \Leftrightarrow x \in \mathit{children}(y)].$$

Because the range of *children'* is *child-set*, the following additional rewrite rule is introduced

$$\forall x \forall y[\mathit{child\text{-}set\text{-}of}(x) = \mathit{children'}(y) \Leftrightarrow x \in \mathit{children'}(y)].$$

These two rules rewrite the statement

$$\forall x \forall y[x \in \mathit{children}(y) \Leftrightarrow x \in \mathit{children'}(\mathit{parent\text{-}set\text{-}of}(y))]$$

as

$$\forall x \forall y[\ \mathit{child\text{-}set\text{-}of}(x) = \mathit{children}(y) \Leftrightarrow \mathit{child\text{-}set\text{-}of}(x) = \mathit{children'}(\mathit{parent\text{-}set\text{-}of}(y))].$$

This completes the classification of *child-set*; the classification of *children'* can now be completed. It is a partial one-to-one function.

Since *children'* was an alternative introduced for *parents*, a cost analysis of both concepts is performed. The estimates come out the same, so for the moment the system records a preference for *parents* (because *parents* is closer to an initial concept than *children'*).

5.6.4 Classification of Children

Now classification of *children* is initiated (*child-set* has already been classified). The concept *children* is a partial function (because child sets can be empty) that is not one-to-one. This places classification at the nodes for partial functions and functions that are not 1–1 (shown shaded in Figure 5.27) where we have been before. Recall that there is an introduction associated with this node. When the range elements of the function being classified are disjoint sets, this rule reflects that range through the function to its domain, creating sets of domain elements that have the same range. The sets of domain elements created in this way are disjoint. The rule also introduces a one-to-one function from the new sets to the original range elements. In this case, the rule can be applied because the range of *children* are disjoint sets. It introduces a sort whose members are sets of parents with the same children. Also it introduces the one-to-one function *parents'* with the following definition,

$$\forall x \forall y[y \in \mathit{parents'}(x) \Leftrightarrow x = \mathit{children}(y)].$$

This is used to rewrite the problem statement

$$\forall x \forall y[\ child\text{-}set\text{-}of(x) = children(y) \Leftrightarrow \\ child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(y))]$$

as

$$\forall x \forall y[\ y \in parents'(child\text{-}set\text{-}of(x)) \Leftrightarrow \\ child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(y))]$$

Recall that earlier the sort *parent-set* was introduced. Members of this sort are sets of parents. When parent sets where found to be disjoint, the partial function *parent-set-of* was introduced, mapping a family member into the *parent-set* he/she is a member of. The rewrite rule derived from the definition of *parent-set-of* now rewrites the statement above as

$$\forall x \forall y[\ parent\text{-}set\text{-}of(y) = parents'(child\text{-}set\text{-}of(x))Leftrightarrow \\ child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(y))]$$

This statement is recognized by classification as defining *parents'* and *children'* as inverses. Recall that earlier the system recorded a preference for *parents* over *children'*. Now the system has determined that *children'* is the inverse of the concept *parents'*. A formulation in terms of both of these is preferable to *parents*. This is the final result of the extended classification of *child*. It was first reformulated in terms of *parents* and *children* and then these two concepts were reformulated as *parents'* and *children'*.

5.6.5 Summary of Extended Classification of Child

This section summarizes the effect of extended classification of *child* on the small FAMILIES problem. The state of the problem after extended classification of *married* is shown in Figure 5.28.

Knowledge acquisition during classification of *child* adds the following statements:

$$\forall x \neg child(x, x)$$

$$\forall \vec{x} \forall \vec{y}[child(\vec{x}, \vec{y}) \Rightarrow \neg child(y, x)]$$

$$\forall x \forall y \forall z[child(x, y) \wedge child(y, z) \Rightarrow \neg child(x, z).$$

Parents and *children* are introduced; knowledge acquisition during classification of *parents* adds the size constraint:

$$\forall x \forall y \forall z \forall w[(y \in parents(x) \ \wedge z \in parents(x) \wedge y \neq z \wedge w \in parents(x)) \\ \Rightarrow w = y \vee w = z]$$

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$couple(P) = couple(Q) \wedge P \neq Q$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[child(x, y) \Leftrightarrow parent(y, x)]$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge x \neq y]$
$\forall x \forall y \forall c[couple(y) = couple(x) \wedge y \neq x \wedge child(x, c) \Rightarrow child(y, c)]$
$\forall x \forall y[couple(x) = couple(y) \wedge y \neq x \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[(couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z) \Rightarrow y = z]$
Query: find-all $x \mid parent(S, x)$

Figure 5.28: State of small FAMILIES after extended classification of *married*

The extended classification continues yielding the problem statement shown in Figure 5.29 and the specialized representation shown in Figure 5.30. The first box in Figure 5.29 encloses the statements captured by `couple`, while the second box encloses those captured by `parent'` and `children'`. The statement preceded by an asterisk is captured in the type hierarchy.

5.7 Deriving New Mixed Constraints

This section discusses how the system finds *implicit* restrictions. Chapter 3 explained that when a concept has an *explicit* restriction on it, a representation is included for that concept even if it is not primitive.[7] For example, the presence of the statement $\forall x \neg brother(M, x)$ in the big FAMILIES problem causes *brother* to be included in the representation even though it is defined in terms of *sibling* and *male*. Representations are included for these concepts because we want restrictions to be reformulated as specific statements. The only concepts that are reformulated are those that get classified and only concepts with representations get classified.

The system identifies explicit restrictions by looking for mixed statements of certain forms in a problem. Clearly this approach will not identify implicit restrictions. For example, the statements

[7]Recall that a restriction is a mixed statement that restricts the number of individuals that can stand in some relation to a specific individual.

$children'(parent\text{-}set\text{-}of(P)) = \{R\}$
$couple(Q) = couple(P) \wedge Q \neq P$
$\exists z[\ \ children'(parent\text{-}set\text{-}of(Q)) = child\text{-}set\text{-}of(z) \wedge$
$\quad children'(parent\text{-}set\text{-}of(z)) = child\text{-}set\text{-}of(S)]$
$\forall x \forall y[child(x,y) \Leftrightarrow parent(y,x)]$
(*) $\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \ \wedge couple(x) = couple(y) \Rightarrow$
$\qquad parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$\forall x\ x \in parent\text{-}set\text{-}of(x)$
$\forall x\ x \in couple(x)$

$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \ \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow$
$\qquad couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$

$\forall x\ child\text{-}set\text{-}of(x) \neq children'(parent\text{-}set\text{-}of(x))$
$\forall x \forall y[\ \ child\text{-}set\text{-}of(y) = children'(parent\text{-}set\text{-}of(x)) \Rightarrow$
$\quad child\text{-}set\text{-}of(x) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z[(child\text{-}set\text{-}of(y) = children'(parent\text{-}set\text{-}of(x)) \wedge$
$\quad child\text{-}set\text{-}of(z) = children'(parent\text{-}set\text{-}of(y))) \Rightarrow$
$\qquad child\text{-}set\text{-}of(z) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z \forall w[\ child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(y)) \wedge$
$\quad child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(z)) \wedge y \neq z \wedge$
$\quad child\text{-}set\text{-}of(x) = children'(parent\text{-}set\text{-}of(w)) \Rightarrow$
$\qquad w = y \vee w = z]$

Query: find-the $parents'(child\text{-}set\text{-}of(S))$

Figure 5.29: Formulation of small FAMILIES after extended classification of *child*.

```
(deftype family-member specializes unique-individual disjoint)
(deftype parent-set specializes
fixed-size-disjoint-set(2,family-member))
(deftype child-set specializes disjoint-set(family-member))
parent-set-of:  partial-function(family-member,parent-set)
child-set-of:  function(family-member,child-set)
parents':  1-1-function(child-set,parent-set)
children':  1-1-function(parent-set,child-set)
parent:  relation(family-member,family-member)
(deftype married-couple specializes
fixed-size-disjoint-set(2,family-member))
couple:  partial-function(family-member,married-couple)
```

Figure 5.30: Representation of small FAMILIES after extended classification of *child*.

$\forall x \neg sibling(M, x)$
$\forall x \forall y [brother(x, y) \Leftrightarrow sibling(x, y) \wedge male(y)]$
imply a restriction on *brother* even though neither is a mixed statement involving *brother*.

The system detects implicit restrictions as problem statements get reformulated to remove explicit restrictions. For example, the presence of the explicit restriction on *sibling* above will cause a reformulation in terms of the function *siblings*, a mapping from an individual to his/her set of siblings. This will cause the second statement above to be rewritten as

$$\forall x \forall y [brother(x, y) \Leftrightarrow y \in siblings(x) \wedge male(y)].$$

When the concept *siblings* is determined to be preferable to *sibling*, the system reformulates the problem in terms of *brothers*. This rewrites the above statement as

$$\forall x \forall y [y \in brothers(x) \Leftrightarrow y \in siblings(x) \wedge male(y)].$$

The idea behind identifying implicit restrictions is that any time a concept is reformulated to remove a restriction (i.e., turned into a specific statement), the system adds restrictions for any concepts that define subsets of the original concept because we know that all subsets of a restricted set will also be restricted. For example, $brothers(x)$ is a subset of the $siblings(x)$.

The system uses one other rule to identify implicit restrictions. When a set of sets $\{s_1, \ldots, s_n\}$ forms a covering of another set S and each a restriction has been identified

on each of the s_i, the system restricts S. For example, In a problem where there are restrictions on brother sets and sister sets, the presence of the statement

$\forall x \forall y[sibling(x,y) \Leftrightarrow brother(x,y) \vee sister(x,y)]$

causes the system to place a restriction on *sibling*. In this case, reformulation and simplification transform this statement into

$\forall x[siblings(x) = brothers(x) \cup sisters(x)]$.

5.8 Detecting Redundant Introductions

It is common for more than one introduction rule to introduce the same concept during design. For instance, recall the example in section 5.3.2 where the concepts *children* and *parents* were introduced for *child*. It turns out that the following introduction rule is associated with the node for functions that are not one-to-one:

> "When the range elements of a function f are sets, i.e., when f is represented as
> `f: function(s1,s2)`
> and `s2` is defined as
> `(deftype s2 specializes set(s3))`,
> introduce the function f' as
> $\forall x \forall y[x \in f(y) \Leftrightarrow y \in f'(x)]$,
> introduce a representation for its range as
> `(deftype F'-ran specializes set(s1))`,
> and represent f' as
> `f': function(s3,f'-ran)`."

When classifying *children*, this rule introduces a concept that is equivalent *parents*; when classifying *parents*, it introduces a concept that is equivalent to *children*. The system will name these equivalent concepts differently and will not know initially that they are equivalent.

In general, detecting that concepts are equivalent is difficult. However, there is an efficient way to determine when the system has introduced a concept that is equivalent to a concept already introduced. A derivation is kept for each introduced concept giving the sequence of introduction rules used to derive it. The system detects the equivalence of two introduced concepts by comparing their sequences.

A set of equivalence reductions is defined over these sequences so that all sequences will be transformed into a unique shortest sequence. This shortest sequence is the canonical member of a class of equivalent sequences starting from the original concept. If two concepts have equal canonical sequences they are equivalent. For example, the derivation given for the version of *children* introduced directly from *child* is,

(left-proj *child*),

where "left-proj" is the name of the rule that introduces a concept *children* with the definition

$\forall x \forall y [x \in children(y) \Leftrightarrow child(y, x)]$.

The derivation of an equivalent concept introduced from *parents* is,

(swap right-proj *child*),

where swap is the name of the rule given earlier in this section and "right-proj" is the rule that introduced *parents* from *child*.

One reduction states that (swap right-proj) should be reduced to (left-proj). This reduces the second derivation of *children* so that it is equal to the first.

Of course, this technique does not help with the more difficult issue of detecting equivalence between two concepts appearing in the initial problem.

5.9 Chapter Summary

This chapter has explained how concept introduction is used to extend classification by introducing new concepts, giving classification new ways to view existing concepts. Introduction rules are attached to nodes in the structure library hierarchies. Classification of a concept collects the rules found at nodes that it reaches. When classification of the concept terminates, the collected rules are applied. They introduce new concepts, defining them in terms of the concept that was being classified. As new concepts are introduced, the system *explores alternative problem formulations.*

If analytical reasoning problems were complete (i.e., contained all the information necessary to solve them), the system would only have to apply introduction rules after classification of a concept when statements mentioning only that concept remained

uncaptured. However, since these problems are incomplete, introductions are always tried because classification of the concepts they introduce can uncover additional missing constraints.

Alternative problem formulations are compared by estimating the cost of capturing the problem constraints in that formulation. Formulations with lower cost estimates are preferred over equivalent formulations with higher cost estimates. The cost of a formulation is the cost of the concepts in it. The cost of a concept is the cost of the machinery capturing the constraints on it. In general, some constraints on a concept are captured by its representation designed by classification and the rest are captured by procedures written by operationalization.

The state of the small FAMILIES problem at the beginning of extended classification is shown in Figure 5.19. The set of represented concepts is $\{married, child, parent\}$. The system performs extended classification on each of these. Extended classification of *married* yields the problem shown in Figure 5.20; the representation produced is shown in Figure 5.21. Extended classification of *child* results in the problem shown in Figure 5.29 and the representation is shown in Figure 5.30. Finally the system considers classification of *parent* but finds that it has already been reformulated as *parents'* and the find-all query transformed into find-the $parents'(child\text{-}set\text{-}of(S))$.

Extended classification of the small FAMILIES problem is now complete. The problem statement and representation passed on to operationalization are those shown in Figure 5.29 and Figure 5.30.

Chapter 6

Operationalization

Operationalization is the representation design system's way to capture the constraints that extended classification fails to capture.

In describing how operationalization captures constraints, it is useful to view a specialized representation as a black box. Inside the box there is a collection of data structures that represent problem situations. The user of the box creates problem situations by "telling" the representation about the specifics of a problem. For example, in creating a situation for the FAMILIES problem, the user "tells" the representation that N is married to P, Q is the grandfather of S, etc.

Operationalization writes procedures that enforce constraints by watching for the execution of tell operations (or combinations of tell operations) that violate a constraint. They respond to the execution of such operations by executing additional tell operations to reestablish the constraint. For example, consider the following constraint:

$$\forall x \forall y [x \in \mathit{siblings}(y) \Leftrightarrow y \in \mathit{siblings}(x)]. \quad (1)$$

One procedure that operationalization writes to capture this watches for the execution of an operation that adds an individual x to the siblings of another individual y. This operation might violate the constraint because it may not be the case that y is in the $\mathit{siblings}(x)$. The procedure reestablishes the constraint by executing an operation to add y to the $\mathit{siblings}(x)$.

As in this example, each constraint left uncaptured after extended classification is

"turned into" one or more procedures that reestablish the constraint. The number of procedures created per constraint is a function of the number of tell operations that can affect the constraint. This is determined from the operations of the data types used to define the representations in the constraint. For example, sibling sets are represented as **sets**. In addition to an operation that adds elements to **sets**, there is another operation that allows all the members of a set to be specified by equating the set to a constant, e.g., $siblings(A) = \{B, C\}$. To fully capture the constraint of (1), operationalization must also write a procedure that responds appropriately when an equate operation is executed.

When procedures have been generated that respond to all operations that can affect a constraint, there is no way to create a situation that violates the constraint, i.e., the constraint is captured.

In general, operationalization captures a constraint in three steps: (i) it determines which representation operations can affect the constraint, (ii) it determines what additional operations should be executed to reestablish the affected constraint, and (iii) it extends the representation with procedures that perform those additional operations whenever one of the affecting operations is executed.

For example, the constraint of (1) is captured as follows. Operationalization first identifies the operations that can affect sibling sets. The representation **sibling-set** is defined in terms of **set**. Suppose that **set** has only two tell operations that get used in building FAMILIES situations: ADD-ELEMENT(y,x) (add x to the set y) and EQUATE-TO-CONSTANT(x,y) (equate the set x to the constant set y).

Operationalization determines the additional operations that must be executed to maintain the constraint of (1) whenever the above operations are executed and then writes two procedures. One procedure responds to the execution of an ADD-ELEMENT(siblings(x),y) by performing an ADD-ELEMENT(siblings(y),x). The other procedure responds to an ASSIGN-TO-CONSTANT(siblings(y),x) by performing an ADD-ELEMENT(siblings(z),y) for each z in the constant set x.

Thus, operationalization converts a general statement into a collection of procedures that respond to the execution of tell operations by executing other tell operations. Since executing a tell operation corresponds to adding a specific fact to a problem

situation, these procedures can be thought of as compiled lemmas that draw conclusions when new specific facts are added to problem situations. These conclusions are additional specific facts that must be added to the problem situation to maintain a constraint.

6.1 Preprocessing for Operationalization

In the previous chapters, figures illustrating the state of the small FAMILIES problem contained statements with named individuals in them. This was done to clearly illustrate the effects of representation design on the problem. As noted, the system, in fact, works with a description of a problem class. The figures in this chapter will use the class description directly. Figure 6.1 shows the class description for the small FAMILIES problem after extended classification. Many, if not all, of the problem statements have been reformulated in terms of representations introduced during extended classification. At this point, the description may contain general statements with existential quantifiers in them such as

$$\forall x \forall y [grandchild(x,y) \Leftrightarrow \exists z (child(x,z) \wedge child(z,y)].$$

$\exists x \exists y [couple(x) = couple(y)]$
$\exists x \exists y [children'(couple(x)) = y)$
$\exists x \exists y \exists z [\ parent\text{-}set\text{-}of(x) = parents(child\text{-}set\text{-}of(z)) \wedge$
$\qquad parent\text{-}set\text{-}of(z) = parents(child\text{-}set\text{-}of(y))]$
$\forall x \forall y [parent\text{-}set\text{-}of(x) \neq \perp \ \wedge couple(x) = couple(y)$
$\qquad \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$\forall x [parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x [couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x \ x \in child\text{-}set\text{-}of(x)$

Figure 6.1: The state of the small FAMILIES class description after extended classification.

It may also contain statements defining concepts that do not have representations. The statement above is also an example of this second type. The previous steps of representation design did not design **grandchild** because *grandchild* is not primitive and does not have any restrictions on it.[1] Finally, statements can contain embedded

[1] Recall that a restriction is a mixed statement restricting the number of individuals that stand

function terms, for example,

$$\exists x \exists y \exists z [\ \textit{parent-set-of}(x) = \textit{parents}(\textit{child-set-of}(z)) \wedge \textit{parent-set-of}(z) = \textit{parents}(\textit{child-set-of}(y))].$$

The class description is preprocessed for operationalization as follows: (i) existential quantifiers appearing in the scope of universals are removed, (ii) definitions of concepts that are not represented are expanded , and (iii) embedded function terms in specific statements are removed.

6.1.1 Removing Existential Quantifiers

The system removes existential quantifiers by skolemization. When a general statement contains an existentially quantified variable[2], the system replaces that variable with a new function of the universal quantifiers in the surrounding scope. For example, the existential z is removed from the statement

$$\forall x \forall y [\textit{grandchild}(x, y) \Leftrightarrow \exists z (\textit{child}(x, z) \wedge \textit{child}(z, y))]$$

by introducing a new function of x and y, $F(x, y)$, and substituting it for z in the statement, yielding

$$\forall x \forall y [\textit{grandchild}(x, y) \Leftrightarrow \textit{child}(x, F(x, y)) \wedge \textit{child}(F(x, y), y))].$$

A representation is also introduced for the new function, e.g., assuming that x is a member of a sort s_1 and that y is a member of a sort s_2:

```
f: function(s1,s2).
```

Problem situations created with statements involving skolem functions refer to unnamed individuals that are denoted by those functions. For example, in a situation created from the statement $\textit{grandchild}(A, B)$, there will be an unnamed individual that is a parent of B and a child of A.

---

in some relation. For example, "A and B are the only grandchildren of C."

[2]Since skolemization is being done on statements in implication normal form, the system must check the sense of a quantified subexpression in the antecedent to determine whether it is existential or universal. For example. in the statement

$$\forall y[(\exists x\ P(x)) \Rightarrow Q(y)],$$

the variable x is universally quantified. To see this, note that the statement is equivalent to

$$\forall y[\neg(\exists x\ P(x)) \vee Q(y)],$$

and to

$$\forall y[(\forall x \neg P(x)) \vee Q(y)].$$

6.1.2 Expanding Definitions

Next defined concepts that do not have representations are identified, all occurrences of such concepts are expanded using their definitions, and then their definitions are removed from the problem. Since these concepts have no representations, it makes no sense to operationalization them in their current form. For example, operationalization would generate a procedure for the definition of *grandchild* which added $grandchild(A, B)$ to a situation in which there was a C such that $child(A, C)$ and $child(C, B)$. But such a procedure makes no sense since there is no representation for *grandchild*. The concept *grandchild* is an example of a concept that is not represented in FAMILIES. Its definition (with existential quantifier removed) is used to expand the statement $\exists x \exists y\ grandchild(x, y)$ into the statement:

$$\exists x \exists y[\ parent\text{-}set\text{-}of(x) = parents(child\text{-}set\text{-}of(F(x,y))) \wedge parent\text{-}set\text{-}of(F(x,y)) = parents(child\text{-}set\text{-}of(y))].$$

Then the definition of *grandchild* is removed from the problem.

6.1.3 Removing Embedded Function Terms

Function terms denote individuals. A specific statement that contains embedded function terms is a special case of a statement that contains individuals in place of those function terms. For example, the presence of the above statement in the small FAMILIES problem implies that problems in that class can contain statements of the form

$$\exists x \exists y[parent\text{-}set\text{-}of(x) = parents(y)].$$

Therefore, embedded function terms are removed from specific statements in developing the class description. The result of doing this for

$$\exists x \exists y[\ parent\text{-}set\text{-}of(x) = parents(child\text{-}set\text{-}of(F(x,y))) \wedge parent\text{-}set\text{-}of(F(x,y)) = parents(child\text{-}set\text{-}of(y))].$$

is

$$\exists x_1 \exists y_1[parent\text{-}set\text{-}of(x_1) = parents(y_1)].$$

The preprocessed description of the small FAMILIES class is shown in Figure 6.2.

$\exists x \exists y\ couple(x) = couple(y)$
$\exists y \exists y[children'(x) = \{y\}]$
$\exists x \exists y\ parent\text{-}set\text{-}of(x) = parents(y)$
$\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \ \wedge couple(x) = couple(y)]$
$\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x\ x \in child\text{-}set\text{-}of(x)$

Figure 6.2: The class description for the small FAMILIES problem.

6.2 The Operationalization Procedure

Operationalization works from the class description, capturing constraints in three steps. In the first step it identifies the *operational literals*[3] in the class description. These are literals that correspond to operations of existing representations. For example, $x \in child\text{-}set\text{-}of(y)$ is an operational literal in the small FAMILIES problem because it corresponds to the ADD-ELEMENT operation of **set** and the class description contains the statement

$$\forall x\ x \in child\text{-}set\text{-}of(x)$$

The system knows which literals correspond to operations because associated with each operation of the library structures is a schema giving the form of the literal that the operation corresponds to. For example, the ADD-ELEMENT operation has the schema $x \in S$ associated with it.

In the second step of operationalization, the operational literals are used to rewrite uncaptured statements until they are sequences of implications involving only operational literals. A statement is rewritten by assuming an operational literal in it, simplifying the statement based on that assumption, and then forming an implication of the form

$$assumption \Rightarrow simplified\text{-}statement.$$

For example, given the operational literal $x \in siblings(y)$, the statement

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

[3] A literal is an atomic formula or a negated atomic formula.

is rewritten by assuming the literal in the statement, yielding

$$true \Leftrightarrow y \in siblings(x).$$

This statement simplifies to $y \in siblings(x)$. Then the following implication is formed:

$$\forall x \forall y [x \in siblings(y) \Rightarrow y \in siblings(x)].$$

More generally, an assumption is made in a statement and then a series of assumptions are made in the final consequent of the statement until what remains is an implication whose antecedent literals are operational and whose final consequent is an unconditional statement involving operational literals.[4] A statement in this form is said to be in *operational form.* Each such sequence is called an *operationalization sequence.*

Recall operationalization captures a constraint by designing a response to every combination of operations that can cause the constraint to be violated. Each statement in operational form corresponds to one combination of such operations, with the consequent corresponding to the appropriate response. If operationalization produces a statement in operational form for every sequence begun from a constraint, the constraint is captured.

The third step of operationalization is to translate the operational forms created by step two into procedures (or daemons) by treating the logical implication $\alpha \Rightarrow \beta$ as,

"When α is true (or becomes true) do β."

For example, the statement

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

is translated into

WHEN ADD-ELEMENT(siblings(y),x) DO ADD-ELEMENT(siblings(x),y)

which is read,

"When x is added to siblings(y), add y to siblings(x)."

This is done by a process that is similar to determining that a literal is operational: each literal is matched against the schemas in the structure library to identify the

[4] An unconditional statement is one that is logically equivalent to a conjunction of literals.

operation that the literal corresponds to.

Note that the procedure generated in the above example does not fully capture the constraint

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

because it does not capture the converse, i.e.,

$$\forall x \forall y [y \notin siblings(x) \Rightarrow x \notin siblings(y)].$$

This is just a special case of the general fact observed above that multiple operationalization sequences are required to fully capture a constraint.

The system handles this case by treating $x \notin siblings(y)$ as a different operation from $x \in siblings(y)$. If a problem contains the operational literal $x \notin siblings(y)$, a separate operationalization sequence is started with the statement

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

which yields:

$$\forall x \forall y [x \notin siblings(y) \Rightarrow y \notin siblings(x)].$$

The next three sections of this chapter detail the three steps of operationalization outlined above.

6.2.1 Identifying Operational Literals

A literal is operational if, when interpreted procedurally, it is an operation supported by the representation in the formula. The system checks each literal appearing in the class description to see if it is operational. It does this by examining the representations of the concepts mentioned in the formula. This check differs depending on whether or not the relation symbol of the formula is a domain relation. When it is, the operations of that relation's representation are checked for a schema matching the literal. The problem statement language has two special relations in it, $\in$ and $=$. All other relations must be domain relations. When a literal's relation symbol is one of the special relations, the system checks the representations of the terms in the argument positions of the formula. For example, in a literal whose relation symbol is $\in$, the representation of the second argument is checked. Thus, $x \in siblings(y)$ is

operational because the range of *siblings* is represented in term of **set**.

Note that a problem's representation may use library types with operations that are never used in building problem situations. The method outlined above for identifying operational literals identifies only operations that are actually used in building problem situations.

Usually the procedural interpretation of an operational literal can either be a tell operation or an operation to test whether something is true of a problem situation. For example, $A \in siblings(B)$ can be interpreted as, "add A to the siblings of B" or as, "test whether A is in the siblings of B." However, some literals can be interpreted only as tests. For example, $A \in \varphi$, where φ is a constant set, can be interpreted as a test, but not as a tell operation because we can not add an element to a constant set. Note that all literals that can be interpreted as tell operations can also be interpreted as tests, but some literals that can be interpreted as tests can not be tell operations.

We will refer to operational literals that can be interpreted only as tests as *test literals*; those with both interpretations will be referred to simply as operational literals. Whether a literal is a test literal or an operational literal can affect the way a procedure is generated for a statement (the third step of operationalization).

Not all literals are operational because the library structures do not have operations for every literal. For example, the statement

$$\forall x \forall y [siblings(x) = child\text{-}set\text{-}of(x) - \{x\}]$$

is not operational because **set** does not support a difference operation. As we will see, non-operational literals are operationalized just as if they were conditional statements.

6.2.2 Operationalization Sequences

The second step in operationalization is to use the operational literals identified in the first step to derive one or more statements in operational form from each uncaptured statement in the class description. As noted, each statement in operational form is derived by one operationalization sequence which repeatedly makes assumptions in a statement until a statement in operational form is produced. This section details this

process for conditional statements and unconditional statements whose literals are not operational. Unconditional statements whose literals are operational are considered to already be in operational form and nothing is done to them here.

The procedure for operationalization sequences works as follows:

1. For each uncaptured statement ϕ and each operational literal that mentions a representation in the statement, begin an operationalization sequence by unifying the operational literals with the literals of the statement. Note that equalities between terms and constant are unified with statements in a special way as discussed below. Also note that equalities between non-constant terms correspond to an operation supported by an equality system that is built into all specialized representations. This is explained in Chapter 7. Assume that there are n operational literals mentioning representations in ϕ and denote these by $\alpha_1, \ldots, \alpha_n$. An operationalization sequence is begun by assuming α_i in ϕ and then deriving an implication for each way that α_i unifies with ϕ. This is done as follows:

   (a) Find all literals in ϕ that unify with α_i. The result of one successful unification is a most general unifier.

   (b) For each unifier, θ, a new statement is constructed whose form is,

   $$\alpha_i[\theta] \Rightarrow \phi[\theta](\alpha_i[\theta]/true).$$

   The notation $\phi[\theta]$ stands for the result of making the substitutions given in θ in ϕ and $\phi(\alpha/\beta)$ is the result of substituting β for every occurrence of α in ϕ. Thus the right hand side of the statement constructed is the result of making the substitutions given in the most general unifier θ in both ϕ and α_i and then substituting *true* for $\alpha_i[\theta]$ in ϕ.

   (c) The resultant statement is simplified according to the rules given in Figure 6.3.

2. If the new statement is in operational form, the sequence is complete. The consequent of the new statement is either (i) an unconditional statement involving only operational literals, (ii) the constant *true*, or (iii) the constant *false*.

$$
\begin{array}{ll}
\neg true \gg false & \neg false \gg true \\
true \wedge P \gg P & false \wedge P \gg false \\
true \vee P \gg true & false \vee P \gg P \\
true \Rightarrow P \gg P & P \Rightarrow true \gg true \\
false \Rightarrow P \gg true & P \Rightarrow false \gg \neg P \\
P \Leftrightarrow true \gg P & P \Leftrightarrow false \gg \neg P
\end{array}
$$

Figure 6.3: Simplification Rules

In case (ii), the new statement is thrown away because a statement of the form

$$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow true)$$

is logically equivalent to *true*. Put in terms of operationalization procedures, the statement can be thrown away because a procedure that adds *true* to a problem situation is useless. In case (iii), the final consequent of the new statement is *false*, i.e.,

$$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow false).$$

Such a statement is replaced by the logically equivalent statement

$$\phi_1 \Rightarrow \cdots (\phi_{n-1} \Rightarrow \neg \phi_n).$$

3. If the new statement is not in operational form, operationalization tries to continue the sequence by making an assumption in the final consequent.

   (a) If no assumptions can be made then operationalization of ϕ fails. In this case, the system fails to capture the constraint of ϕ in the designed representation. A representation that partially captures a problem's constraints can be used as a specialized reasoner working along side a theorem prover to solve problems. The theorem prover need consider only those statements that are not captured, leaving the constraints of the other statements to be enforced by the specialized representation.[5]

   (b) If it finds an assumption in β in the statement,

   $$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \cdots (\alpha_n \Rightarrow \beta))$$

   then the sequence is continued with,

[5] I have not attempted to fully consider interfacing a specialized reasoner with a theorem prover in my research. There is a body of research on just this topic (e.g., [Miller & Schubert 88]).

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow ...(\alpha_n \Rightarrow (\alpha_{n+1} \Rightarrow \beta\prime)))),$$

where α_{n+1} is the new assumption and $\beta\prime$ is the result of substituting *true* for α_{n+1} in β.

As an example of this process, consider the statement

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

and the operational literal $x_1 \in siblings(y_1)$.[6]

An operationalization sequence is started by assuming $x_1 \in siblings(y_1)$ in the above statement. This is done by finding unifications of the literal with literals of the statement. One such unification results in,

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow (true \Rightarrow y_1 \in siblings(x_1))],$$

which is simplified to,

$$\forall x_1 \forall y_1[x_1 \in siblings(y_1) \Rightarrow y_1 \in siblings(x_1)].$$

This statement is in operational form, completing the first sequence.

Another operationalization is begun with the other possible unification of $x_1 \in siblings(y_1)$ with a literal of the original statement, i.e., $y \in siblings(x)$. This results in the same statement as the first sequence.

The above procedure is modified slightly when an assumption is an equality with a constant term, e.g., $siblings(x) = \varphi$. Instead of unifying the assumption with literals of a statement, unifications are found between the non-constant term ($siblings(x)$) and terms of the literal in the statement being operationalized. Then for each unification, the simplified form of the statement's consequent has φ substituted for $siblings(x)$. Thus, for each unification θ found, the new formula generated for a statement of the form,

$$\alpha_1 \Rightarrow (\alpha_2 \Rightarrow \cdots(\alpha_n \Rightarrow \beta)),$$

has β replaced by,

$$F(x) = \varphi[\theta] \Rightarrow \beta(F(x)[\theta]/\varphi).$$

We continue the example from above with the operational literal $siblings(x_2) = \varphi$,

[6]Variables with subscripts are used in these examples to make it clear when variable substitutions have occurred.

assuming it in

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

This is done by unifying $siblings(x_2)$ with terms of its literals, then substituting φ for $siblings(x_2)$. One term it unifies with is $siblings(y)$. The result of substituting x_2 for y and $siblings(x_2)$ for $siblings(y)$ is

$$x \in \varphi \Leftrightarrow x_2 \in siblings(x).$$

The new statement created in this step is

$$\forall x \forall x_2 [siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Leftrightarrow x_2 \in siblings(x))]. \quad (2)$$

This statement is not in operational form so operationalization looks for an assumption to make in the consequent. Suppose it assumes $x \in \varphi$, deriving the formula

$$\forall x \forall x_2 [siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Rightarrow (True \Leftrightarrow x_2 \in siblings(x)))],$$

which is simplified to

$$\forall x \forall x_2 [siblings(x_2) = \varphi \Rightarrow (x \in \varphi \Rightarrow x_2 \in siblings(x))].$$

Next the system assumes $x_3 \in siblings(x_4)$ in (2). This yields the operational form:

$$\forall x_4 \forall x_3 [siblings(x_4) = \varphi \Rightarrow (x_3 \in siblings(x_4) \Rightarrow x_4 \in \varphi)].$$

Since $siblings(x_2)$ also unifies with the term $siblings(x)$ in the statement

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

operationalization finds another way to assume $siblings(x_2) = \varphi$ in the statement. This sequence results in the same operational form as the last sequence.

When an operationalization sequence produces a statement whose final consequent is a non-operational literal, it continues to make assumptions in an effort to simplify it. That is, it continues just as though the final consequent of the statement were conditional. Unconditional general statements whose literals are not operational are also dealt with in this way. For example, the FAMILIES problem class description contains the statement:

$$\forall x [siblings(x) = child\text{-}set\text{-}of(x) - \{x\}].$$

This is operationalized like any conditional general statement. Suppose, for instance, that the problem's class description contains the statement

$\exists x_1\ siblings(x_1) = \emptyset$,

making $siblings(x_1) = \emptyset$ an operational literal. An operationalization sequence is started by assuming this literal in the statement, yielding

$siblings(x_1) = \emptyset \Rightarrow child\text{-}set\text{-}of(x_1) - \{x_1\} = \emptyset$.

The next section explains that statements generated during operationalization are simplified. This statement is simplified by the mechanism described there to

$siblings(x) = \emptyset \Rightarrow child\text{-}set\text{-}of(x) = \{x\}$,

which is in operational form.

Rewriting Intermediate Statements

As noted, statements containing the constants *true* and *false* are generated during operationalization and then simplified according to the rules given in Figure 6.3. In addition, the intermediate statements are subjected to a body of rewrite rules that attempt to further simplify them based on mathematical knowledge, mainly about sets. Simplifying rewrites are applied iteratively to statements until no more are possible. Here is an example of how the rewrite system is integrated with the operationalization process. Suppose an class description contains the statements

$\forall x \forall y [x \in brothers(y) \Leftrightarrow x \in siblings(y) \wedge sex(x) = male]$

$\exists z\ siblings(z) = \emptyset$.

The system begins an operationalization sequence by assuming $siblings(z) = \emptyset$ in the general statement. This results in the statement

$\forall x \forall z [siblings(z) = \emptyset \Rightarrow (x \in brothers(z) \Leftrightarrow x \in \emptyset \wedge sex(x) = male)]$.

There is a rewrite rule that embodies the fact that empty sets have no elements by simplifying the literal $x \in \emptyset$ to *false*. This rule rewrites the above statement as

$\forall x \forall z [siblings(z) = \emptyset \Rightarrow (x \in brothers(z) \Leftrightarrow false \wedge sex(x) = male)]$,

which is then simplified to

$\forall x \forall z [siblings(z) = \emptyset \Rightarrow x \notin brothers(z)]$.

Another rewrite, which embodies the fact that sets with no elements are equal to the empty set, is applied to this statement; this yields

$\forall x \forall z [siblings(z) = \emptyset \Rightarrow brothers(z) = \emptyset]$,

which is recognized as being in operational form.

In general, the success of operationalization can depend on the simplification rules that the system has. A more complete discussion of the rewrite system, including its current collection of rules is given in appendix A. This collection has proven sufficient for the twelve problems that the system has been tested on.

Optimizations of Operationalization

Operationalization is expensive. To a certain extent this does not matter because the cost of operationalization is incurred at representation design time, not at problem solution time. This is especially true if the resultant representation will be used repeatedly because the cost of operationalization can be amortized.

One reason operationalization is expensive is that it can generate a large number of procedures to operationalize a statement. This does effect the efficiency of the representation designed. The system has several strategies for reducing the cost of operationalization. For the twelve problems the system has been tried on so far, these strategies allow a representation to be designed and used to solve the problem in much less time than a theorem prover using brute force search in the original representation.

One strategy that the system uses is to try to minimize the use of operationalization at all, by capturing the constraints of statements during extended classification. A second strategy is to make operationalization more efficient. One technique is to reduce the number of sequences generated for a particular statement. Other efficiency-enhancing techniques detect when operationalization produces statements that are subsumed by existing statements in operational form, and in that case avoids generating a procedure for them.

The technique that reduces the number of operationalization sequences exploits the fact that separate statements with the same final consequents are equivalent if their sequences of antecedents are permutations of each other. For example, the following two statements are logically equivalent:

$P \Rightarrow (Q \Rightarrow R)$
$Q \Rightarrow (P \Rightarrow R)$.

It is also the case that the procedures generated for these two statements are equivalent. This fact is used to avoid operationalizing statements with all combinations of literals, reducing the number of operationalization sequences generated for a statement.

The system currently has one technique to detect when it has generated a statement that is logically subsumed by another statement in operational form. One way to increase the efficiency of operationalization and of the representations the system designs is to add more techniques of this type.

The existing technique exploits the fact that if a procedure is included for a statement, then including a procedure for a less general statement will not change the behavior of a representation. It removes any statement of the form

$$\phi_1 \Rightarrow \cdots(\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m))$$

if operationalization has generated a statement whose consequent is a conjunction of literals which is a superset of $\{\psi_1, \ldots, \psi_m\}$ and whose antecedents are a subset of $\{\phi_1, \ldots, \phi_n\}$. For example,

$$P \Rightarrow (Q \wedge R)$$

subsumes

$$S \Rightarrow (P \Rightarrow R).$$

To see that any statement of the second form will subsume any statement of the first, consider the following two step argument. First, as has been noted, the statement

$$\phi_1 \Rightarrow \cdots(\phi_n \Rightarrow \psi)$$

is logically equivalent to

$$(\phi_1 \wedge \cdots \wedge \phi_n) \Rightarrow \psi.$$

In this second form it is clear that any statement obtained by removing literals from the conjunction in the antecedent is more general than this statement because it concludes the consequent from fewer conditions. Now consider the following two statements:

$$\phi_1 \Rightarrow \cdots(\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)) \quad (3)$$

$$\phi_1 \Rightarrow \cdots(\phi_{n-1} \Rightarrow (\gamma_1 \wedge \cdots \wedge \gamma_{m'})). \quad (4)$$

From the previous step of the argument we know that if

$\{\psi_1, \ldots, \psi_m\} = \{\gamma_1, \ldots, \gamma_{m'}\}$,

then (4) is more general than (3). Notice that this is still the case if

$\{\psi_1, \ldots, \psi_m\} \subseteq \{\gamma_1, \ldots, \gamma_{m'}\}$

because all of the conclusions of (3) are also conclusions of (4).

6.2.3 Generating Procedures from Statements in Operational Form

This section first explains the process of generating procedures for unconditional statements, then for conditional statements. As a result of the last step, all statements in the class description are in operational form. An unconditional statement made up of operational literals is in operational form. An unconditional general statement is compiled into a daemon that responds to the creation of new individuals of the sort found in the statement. Its response is to execute operations corresponding to the literals in the statement. For example, the procedure generated for the statement

$\forall x \; x \in couple(x)$

responds to the creation of a new married couple by adding an element to it. For instance, if a situation does not contain a data structure corresponding to $couple(B)$, then adding a statement containing that term will cause such a data structure to be created. When this happens, the procedure that enforces the constraint of the above general statement executes the operation ADD-ELEMENT(couple(B),B).

Recall that the specific statements of the class description are derived from the specific statements of the original problem by replacing the individuals with existentially quantified variables. There is no point in translating the specific unconditional statements of the class description into procedures because we are interested in the actual problem situation. Instead we translate the original specific statements. For example, the statement $couple(Q) = couple(P)$ is translated into code that creates a couple containing P and Q and this code is executed in creating the problem situation.

An unconditional specific statement is translated into a sequence of operations. For example, the statement

$A \in siblings(B) \wedge B \in siblings(C)$

is translated into

ADD-ELEMENT(siblings(B),A)
ADD-ELEMENT(siblings(C),B).

Conditional statements are compiled into daemons that respond when operations corresponding to their antecedents are executed. The basic idea is to compile a statement of the form,

$$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)),$$

where all the ϕ_i and ψ_i are operational literals, into a sequence of daemons. Each daemon in the sequence, except the last, waits for the execution of an operation and then creates a new daemon. The last daemon, e.g., the daemon corresponding to $\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)$ above responds to the execution of an operation corresponding to its antecedent ϕ_n by executing an operation corresponding to each literal in the consequent.

Suppose that the operation corresponding to each of the ϕ_i above is Φ_i and that the operation corresponding to each of the ψ_i is Ψ_i. Then the basic form of the procedure generated for the statement above is,

WHEN Φ_1 DO
    WHEN Φ_2 DO
        ⋮
        WHEN Φ_n DO $\Psi_1, \ldots, \Psi_m$

The execution of the statement "WHEN op DO ..." causes a daemon to be created that waits until an operation (op) is executed and responds by executing the statements after the DO. When an operation Φ_1 is executed, the above procedure executes a statement of the form,

WHEN Φ_2 DO
    ⋮
    WHEN Φ_n DO $\Psi_1, \ldots, \Psi_m$

This creates a new daemon because the bindings for the arguments of Φ_1 are substituted into $\Phi_2, \ldots, \Phi_n$. As a concrete example, consider the statement

$x \in siblings(y) \Rightarrow (sex(x) = male \Rightarrow x \in brothers(y))$.

The procedure generated for this statement is,

```
WHEN ADD-ELEMENT(siblings(y),x) DO
        WHEN ASSIGN-CONSTANT(sex(x),male) DO
                ADD-ELEMENT(brothers(y),x)
```

When a particular operation is executed that adds an individual to the siblings of another individual, say ADD-ELEMENT(siblings(B),A), this daemon responds by executing the following statement:

```
WHEN ASSIGN-CONSTANT(sex(A),male) DO
        ADD-ELEMENT(brothers(B),A)
```

This creates a new daemon that waits for the execution of the operation ASSIGN-CONSTANT(sex(A),male).

Note that WHEN does not create daemons like those in a standard object oriented programming language. A standard way to implement a daemon like the one in the above example is to produce code that executes after an ADD-ELEMENT message is sent of a sibling set object. The problem with this approach is that in specialized representations the object denoted by siblings(A) may have other names too, making it possible for an element to be added to it without explicitly referring to it as siblings(A). Therefore, there must be a method of identifying the names of an object when an operation is performed on it. The method that specialized representations use involves the integration of daemons with an equality system. This is explained in Chapter 7.

To completely capture the constraint of a statement, the procedures generated for it must work independently of the order of satisfaction of the antecedent literals. One way to accomplish this is to allow operationalization to generate statements for all permutations of a statement's antecedents. This approach is too expensive: if a statement being operationalized has n antecedents, then $n!$ statements are created to operationalize it.

As noted earlier, the system has a more efficient approach in which the procedures

generated are insensitive to the order in which the operations of a statement's antecedents are executed. For example, the procedure generated for the statement

$$x \in siblings(y) \Rightarrow (sex(x) = male \Rightarrow x \in brothers(y))$$

works even if the operation ASSIGN-CONSTANT(sex(A),male) is executed before ADD-ELEMENT(siblings(B),A).

This is accomplished by compiling all but the first literal so that the procedure checks to see if the antecedent is true before generating a daemon that waits for it to become true. This check is generated from the test interpretation of the operational literal. For instance, the procedure generated for the example above involving *siblings* and *brothers* is actually compiled into

```
WHEN ADD-ELEMENT(siblings(y),x) DO
        IF sex(x)=male THEN ADD-ELEMENT(brothers(y),x)
        ELSE WHEN ASSIGN-CONSTANT(sex(x),male) DO
                        ADD-ELEMENT(brothers(y),x)
```

The expression sex(x)=male is a test operation supported by the representation of *sex*.

One additional complication arises in the compilation process. Sometimes when constants get substituted for variables in a statement, the result is literals which no longer have tell interpretations. For example, recall the operationalization of the statement

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

given the literal $siblings(y_1) = \varphi$ (where φ is a constant). One statement generated is

$$\forall x \forall y[siblings(y_1) = \varphi \Rightarrow (y_1 \in siblings(x) \Rightarrow x \in \varphi)].$$

We can test whether an individual is a member of a constant set, but there is no operation that adds individuals to constant sets.

This complication is straightforward to address. As explained above, literals are usually compiled into a test followed by a daemon that waits for a corresponding operation. Literals that correspond only to tests are compiled into tests with no following WHEN statement. For example, the literal $x \in \varphi$ in the above statement

is compiled into a test to make sure that the constant bound to x is a member of φ, i.e., signal a contradiction if its not.

Some test literals are compiled into iteration constructs. For example, another statement generated in operationalizing

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

is

$$\forall x \forall y [siblings(y_1) = \varphi \Rightarrow (x \in \varphi \Rightarrow y_1 \in siblings(x))].$$

Note that unlike the example above, the first occurrence of x is in a test literal. Since this does not correspond to an operation and since this is the first occurrence of x, it will be unbound when it comes time to perform the test. Because of this the literal $x \in \varphi$ is compiled into the program fragment

FOR EACH $x \in \varphi$ DO...

In the previous example, the test literal is treated differently depending on whether its arguments are bound to constants at the time the test is performed. This is a special case of the following phenomenon: The ability to generate a test at all for a literal depends on the whether its arguments are bound when the test is performed. For example, consider the statement

$$\forall x \forall y \forall z [x = spouse(y) \Rightarrow (y = spouse(z) \Rightarrow x = z)].$$

The procedure generation process for this statement begins with

WHEN ASSIGN-CONSTANT(spouse(y),x) DO...

In the next step, the system tries to generate a test for the literal $y = spouse(z)$ and notes that z is unbound. No test can be generated for this because it requires checking the spouses of all individuals.

The system attempts to deal with this problem by reordering the antecedents of a statement and trying to generate a procedure for the new statement. Continuing the above example, when the system finds that it can not generate a test for $y = spouse(z)$, it changes the above statement to

$$\forall x \forall y \forall z [y = spouse(z) \Rightarrow (x = spouse(y) \Rightarrow x = z)].$$

This time when the second antecedent is reached, y is bound and a test can be

generated. [7] The procedure produced for this statement is

```
WHEN ASSIGN-CONSTANT(spouse(z),y) DO
        IF x=spouse(y) THEN
           IF x≠ z THEN contradiction
        ELSE WHEN ASSIGN-CONSTANT(spouse(y),x) DO
                    IF x≠ z THEN contradiction.
```

6.3 Soundness of Operationalization

Problem situations are built by executing the tell operations of a representation to add the specific facts of the problem. The data structures built always represent a set of specific facts. The *information content* of a situation is the set of specific facts that its data structures represent.

To show that operationalization is sound, we must show that the procedures produced for a statement ϕ only add facts to problem situations that follow from ϕ. More precisely, let P be a procedure generated in operationalizing ϕ and let Φ be the information content of a problem situation at the time P is executed. Then we must show that the facts added to Φ by P follow from the set of statements $\{\Phi, \phi\}$.

We show this in two steps. First, we show that for any statement ψ produced in operationalizing ϕ, $\phi \vdash \psi$ (i.e., ψ follows from ϕ). Then, we argue that, as a consequence, the procedure generated from ψ must add facts to a situation that follow from ϕ.

Consider the process by which ψ is generated from ϕ. In the first step, an assumption α is made in ϕ and the statement

$$\alpha \Rightarrow \phi'$$

is produced. The expression ϕ' in this statement is the result of substituting all occurrences of α in ϕ by *true* and simplifying. It is easy to show that

$$\phi \Rightarrow (\alpha \Rightarrow \phi').$$

[7] Notice that we could keep a list of all the individuals in the problem and then it would have been possible to operationalize the original form of this statement. The result would be a less efficient procedure. Thus, in this case, we got lucky. This illustrates the current method of reordering antecedents is not particularly robust. See [Smith85] for a more general discussion of antecedent reordering.

We prove this statement by contradiction. Suppose to the contrary that ϕ is true but that $\alpha \Rightarrow \phi'$ is false. For this to be so, α must be true and ϕ' must be false. But because of the way ϕ' was derived from ϕ, if α is true, ϕ is the same as ϕ'. This contradicts our assumption that ϕ and ϕ' have different truth values.

Since ψ is the result of zero or more applications of the assumption making process, $\phi \vdash \psi$.

Suppose that ψ has the form

$$\phi_1 \Rightarrow \cdots (\phi_n \Rightarrow (\psi_1 \wedge \cdots \wedge \psi_m)).$$

The procedure P_ψ, generated from ψ, will add the facts $\psi_1, \ldots, \psi_m$ to situations containing the facts

$$\phi_1 \wedge \cdots \wedge \phi_n.$$

Thus if P_ψ executes in situation Φ, it will add facts that follow from $\{\Phi, \psi\}$ and, since we have already shown that $\phi \vdash \psi$, it follows that P_ψ will only add facts that follow from $\{\Phi, \phi\}$.

We would also like operationalization to be complete, i.e., we would like the procedures produced for a statement ϕ add *all specific* facts to problem situations that follow from ϕ. More precisely, let Φ be the information content of a problem situation and let Ψ be any subset of Φ. We would like it to be the case that if ϕ has been successfully operationalized and a specific fact σ follows from the set of statements $\{\Psi, \phi\}$, then σ appears in Φ.

However, operationalization is not complete. One source of the incompleteness is that the representations designed by the system can build only very limited kinds of disjunctive situations. In general, disjunction in a problem situation can not be represented. The difficulty is best illustrated by an example. Consider the following propositional problem.

P
$P \Rightarrow (W \vee T)$
$P \Rightarrow (R \vee S)$
$(W \wedge R) \Rightarrow Q$
$(W \wedge S) \Rightarrow Q$
$(T \wedge R) \Rightarrow Q$
$(T \wedge S) \Rightarrow Q$
Query: $\Box Q$.

Q follows in this problem. However, consider a representation capturing the conditional statements of this example by operationalization. If we use it to create a situation representing P, this situation will not contain Q. The difficulty is that there is no way to create a problem situation representing $W \vee T$ without committing to one of them actually being true. More precisely, to capture

$$P \Rightarrow (W \vee T), \tag{5}$$

operationalization includes procedures for the operational forms

$P \Rightarrow (\neg W \Rightarrow T)$ and
$P \Rightarrow (\neg T \Rightarrow W)$

(among others). These are the only procedures included for (5) computing consequences of P. Therefore, when P is added to a problem situation in this representation, no additional facts will be added to the situation.

6.4 Summary

This chapter has explained the process of operationalization, proven it to be sound, and demonstrated that it is incomplete when arbitrary disjunctive situations must be represented to solve a problem. Operationalization is the representation design system's way to capture the constraints that extended classification fails to capture. It begins with the statements left uncaptured by extended classification. Figure 6.2 shows the statements left uncaptured by extended classification of the small FAMILIES problem.

Operationalization of small FAMILIES begins with the following operational literals:

$couple(x) = couple(y)$
$children'(x) = \{y\}$
$parent\text{-}set\text{-}of(x) = parents(y)$
$x \in child\text{-}set\text{-}of(y)$.

Operationalization proceeds by assuming these literals in the appropriate general statements, trying to derive statements in operational form.

Literals of the form $F(x) \neq \perp$ are treated specially in this process. Recall that this states that x has an image under the partial function F. During operationalization, if the class description contains any operational literals of the form $F(x) = y$ (for any arbitrary term y) then there can be some individuals that have images under F. Therefore, if the literal $F(x) \neq \perp$ appears anywhere in the class description, $F(x) \neq \perp$ is made an operational literal of the problem class.

For example, the statements

$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\exists x \exists y\ couple(x) = couple(y)$

in the small FAMILIES problem cause the statement

$\exists x\ couple(x) \neq \perp$

to be added. As with any other operational literal, this causes an operationalization sequence to be started by assuming $couple(x) \neq \perp$ in

$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$

which results in this same statement.

Literals of the form $F(x) \neq \perp$ are compiled into daemons that respond to the creation of range elements of the function F. For example, the procedure generated for the statement above responds to the creation of any $couple(x)$ by adding x to it.

The procedure generated for the statement

$\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$

responds to an operation that makes two individual's parent sets the same by making their couples the same. [8]

The procedure generated for the statement

$$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \ \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$$

does the following. When a parent set is created for an individual x who is an element of a couple, the parent set of the other individual in that couple is equated with the

[8] Chapter 7 explains what it means to make to structures like parent sets and couples the same.

new parent set.

Code is also generated for the specific statements containing named individuals. Execution of this code causes the situation representing the small FAMILIES problem to be created. This process is described in Chapter 7.

Chapter 7

Representation Machinery

This chapter details how the library types are implemented and how representations designed in terms of them are used to solve problems. It explains three underlying mechanisms that the representations rely on in building problem situations. It explains how the library types `relation`, `function`, `set`, and `individual` are implemented. Finally, it shows how the representation designed for the small FAMILIES problem is used to build one problem situation.

Recall that library structures are implemented as parameterized abstract data types (ADTs). Each library ADT is a prototype for creating representations. Concept representations are created from ADTs in the concept hierarchy by instantiation. For example, an instance of type `relation` is used to represent *married*. This structure is created by the declaration

```
married: relation(family-member,family-member).
```

Representations of sorts are created from ADTs in the sort hierarchy by defining subtypes. For example, a representation of the sort *family-member* is created by the definition

```
(deftype family-member specializes individual disjoint).
```

Instances of the ADT `family-member` are used to represent family members like *N*.

7.1 Underlying Mechanisms

Three facilities support specialized representations: an equality system, a mechanism that creates anonymous individuals, and a daemon invocation mechanism. These are closely integrated in the process of creating situations, so they are described together.

7.1.1 The Equality System and Anonymous Individuals

The equality system is similar to RUP[McAllester82]. It maintains equivalence classes of terms known to be equal and supports an operation that merges equivalence classes as new terms are found to be or asserted to be equal.

The implementation distinguishes *terms*, which are linguistic items, from *objects*, which represent semantic items, i.e, things in the problem domain. In this chapter, we will indicate that an item is a term using the italic font (e.g., A) and that it is an object using typewriter font (e.g., `A`).

Unlike RUP, our equality system requires that each equivalence class have exactly one object associated with it that is the object denoted by the terms of this class. For example, in the situation representing

$$mother\text{-}of(A) = B, mother\text{-}of(C) = mother\text{-}of(A),$$

the terms $mother\text{-}of(A)$ and $mother\text{-}of(C)$ are placed in the same equivalence class. The object associated with this class is `B`.

The object denoted by an equivalence class may be unknown at the time the class is created. For example, a situation representing only the statement

$$mother\text{-}of(C) = mother\text{-}of(A),$$

has an equivalence class containing the terms $mother\text{-}of(C)$ and $mother\text{-}of(A)$, but the object denoted by this class is unknown. In cases like this one, the system creates an anonymous object and associates it with the class. The type of the anonymous object (i.e., the sort of the individual it represents) is determined from the range of *mother-of* to be $family\text{-}member$.

The equality system also allows two equivalence classes to be marked to indicate that

they are known to be disjoint (i.e., the individuals that the two classes denote are known to be unequal). Any attempt to equate individuals in classes so marked results in the equality system signalling a contradiction.

When the equality system merges two equivalence classes, it creates a new class which is the union of the terms in the two original classes. Since each class denotes exactly one object, merging two classes requires equating the objects associated with them. To determine what object the new class denotes, the system sends a signal to the object associated with one of the original classes notifying it that it is being equated with the other object. This is called an EQUATE notification. When two objects are sent an EQUATE notification, they respond with a single object which the equality system associates with the new class.

Accordingly, the library ADTs used to create objects are required to respond to the EQUATE notification. In the current library, the ADTs that do this are **individual**, **set**, and their specializations.

To illustrate the merging process, consider the situation created to represent

$$mother\text{-}of(A) = B,\ mother\text{-}of(C) = D.$$

It will contain two equivalence classes: one containing the term $mother\text{-}of(A)$ with the associated object B and the other containing the term $mother\text{-}of(C)$ with the associated object D. Now suppose the statement $mother\text{-}of(A) = mother\text{-}of(C)$ is added. The two classes are merged and the objects `B` and `D` are sent an EQUATE signal. They respond with a single object that can be referred as B or D (i.e., $B = D$).

7.1.2 The Daemon Invocation Mechanism

Section 6.2.3 explained how operationalization generates daemons for conditional statements that are not captured by classification. These can not be implemented in the way daemons are standardly implemented in object oriented programming languages. Consider an example of the difficulty. One of the daemons operationalization generates for the statement

$$\forall x \forall y [x \in child\text{-}set\text{-}of(y) \wedge x \neq y \Rightarrow x \in siblings(y)]^1$$

[1] Recall that the *child-set* of an individual is a set of individuals with the same parents as the

is

WHEN ADD-ELEMENT(child-set-of(y),x) DO
IF $x \neq y$ THEN ADD-ELEMENT(siblings(y),x).

A standard way to implement a daemon like this one is to produce code that executes after an ADD-ELEMENT message is sent to a `child-set` object, conditionally adding an element to a `sibling-set` object. The problem with this approach is that, in specialized representations, the object denoted by *child-set-of*(A) may have other denotations (i.e., other terms denoting it), making it possible for an element to be added to it without explicitly referring to it as *child-set-of*(A). For example, suppose

child-set-of(A) = *child-set-of*(B).

If we add an element to *child-set-of*(B), the standard implementation of the above daemon will only add that element to *siblings*(B).

In light of this difficulty, our daemon invocation mechanism makes use of the equivalence classes maintained by the equality system. Daemons are attached to terms rather than to objects. When an operation is performed on an object, the system searches through the terms denoting that object, invoking daemons that respond to the operation performed. For example, when an element is added to the object denoted by *child-set-of*(B), the system searches the equivalence class of terms denoting that object. It finds the terms *child-set-of*(A) and *child-set-of*(B) and invokes the above daemon on both.

Note that accessing terms in this way is also how the system determines the references for variables in daemons. Thus, in the above example it is from the terms *child-set-of*(A) and *child-set-of*(B) that bindings of `A` and `B` are established for y in the daemon.

Invocation becomes more complicated when daemons have nested patterns in their WHEN conditions. Consider, for example, a daemon like

WHEN op(F(G(x))) DO ...

The following difficulty arises: When the operation op is executed, its argument will be an object, say `A`. In response, the system looks for terms denoting `A` of the

given individual.

form $F(y)$. The above daemon is associated with terms of this form. However, the daemon should not be invoked on all terms of the form $F(y)$, only on those in which y is an object denoted by terms of the form $G(x)$. As the nesting level in a daemon increases, the process of filtering out those terms to which a daemon should be applied involves more levels of "indirection." This can get quite expensive. Fortunately, nested general statements are rare in analytical reasoning problems and, consequently, so are nested daemons.

7.2 Implementation of Library ADTs

This section describes how the library ADTs `individual`, `relation`, `function`, and `set` are implemented. Except for `1-1 function`, the rest of the library types are implemented as specializations of these. The implementation of `1-1 function` is also described below.

The ADT `individual` is used to represent domain individuals. It is used as a prototype to create representations of sorts by defining subtypes such as

```
(deftype family-member specializes individual disjoint).
```

`Individual` is able to represent the fact that an individual can have more than one name. It does this with a single data field, called **name**, which is used to store a list of constants that name the individual represented by an instance. For example, the instance representing the individual B has the name field (B). `Individual` has several procedures associated with it (answers several messages). When an instance of individual receives an EQUATE message of the form

EQUATE(other-object)

it responds by returning a single instance whose name field is the union of its name field and other-object's name field. For instance, in the situation representing

$mother\text{-}of(A) = B$
$mother\text{-}of(C) = D$

there are objects representing each of the individuals B and D. When the the statement $mother\text{-}of(A) = mother\text{-}of(C)$ is added to this situation, one of these objects, say `B`, is sent the message

EQUATE(D).[2] The response to this message is to combine the objects representing B and D and return the single object whose name field contains the list (B D).

Individual also supports an EQUAL? message of the form

EQUAL?(other-object).

When an instance receives such a message, it responds with *true* if its name field shares any elements with other-object's name field. If the instance and other-object are associated with equivalence classes that are marked disjoint, the instance responds with *false*. Otherwise it responds with *unknown*.

In the FAMILIES problem, the different names given for individuals of sort *family-member* stand for different individuals. Therefore, the collection **family-member** is specialized to **unique-individual**. This ADT also has a **name** field, however, it can only contain a single name. An instance of **unique-individual** responds to an EQUAL? message with *true* if the names are the same or they are anonymous and associated with the same equivalence class; *false* if they have different names or are associated with equivalence classes that are marked disjoint; and *unknown* otherwise.

An instance of **unique-individual** responds to an EQUATE message as follows. If both instances are anonymous or they have the same name, it does not matter which one is returned so the receiving instance returns itself. If one of the individuals is anonymous and the other is named, then the named individual is returned. If both individuals are named and the names are different, a contradiction is signalled.

Note that the effect of equating an anonymous individual with a named individual is to name the anonymous individual. In a typical scenario, as situations get created anonymous individuals get named.

The ADT **relation** is implemented as two lists of ordered n-tuples. Instances of **relation** are created to represent particular relations. An instance R created for a relation R contains two lists used to store n-tuples of individuals. One list in R is used to store the n-tuples of individuals known to stand in the relation R. The other list is used to store n-tuples of individuals known not to stand in the relation R. As

[2] Recall that D is the object representing D.

a problem situation is created n-tuples get added to these lists.

Instances of the ADT **function** are used to represent functions. Unlike **relations**, instances of **function** do not store all pairs in a central list, i.e., the is no list associated with a **function**, say **mother-of**, storing pairs of the form **<x,mother-of(x)>**. Instead each individual that has an image under a function has a pointer to its image element. The pointer is labeled with the function name. Thus, $A = mother\text{-}of(B)$ is represented by the object representing B having a pointer to the object representing A labeled **mother-of**. The single valued property of functions is enforced as follows. Each individual may have several function pointers (may have images under several functions), but may have only one pointer labeled with each function's name. If there is an attempt to make more than one object be the image of some object under a function, the system attempts to equate the two objects. This may cause a contradiction.

Function terms can appear in problem statements that do not give names for the individuals they denote. Creating situations for these statements requires creating anonymous (unnamed) individuals. For example, the statement

$$married(mother\text{-}of(A), father\text{-}of(B)),$$

is represented by creating an anonymous family member, say **x**, to stand for the $mother\text{-}of(A)$ and another anonymous individual, say **y**, to stand for the $father\text{-}of(B)$. Then the system adds the pair **<x,y>** to **married**.[3]

Technically the ADT **1-1 function** is not implemented as a specialization of **function**. Instead, the system creates an inverse for each **1-1 function**. This can be named by a problem statement; if not, the system generates a name. Then whenever a functional relationship is added to a problem situation, if the function has an inverse, the inverse relationship is also added. This allows the existing mechanism enforcing the single valued property to also enforce the one-to-one property.

The ADT **set** is instantiated to represent individual sets and subtypes are defined from it to represent sorts. As an example of its use to represent a sort, consider *brother-set*, the sort of sets of brothers of the same person. It is represented with a subtype whose instances are **sets** of brothers. **Set** has a three data fields: one

[3]This assumes, of course, that *married* is represented as a relation.

for storing a list of the individuals known to be members of a particular set, one for storing individuals that are known not to be members of a particular set, and one to indicate whether a set is closed (i.e., all the members of that set are known). It has the following procedures associated with it:

1. ADD-ELEMENT adds a new individual to a set unless the set is closed. In this case a contradiction is signalled unless the set already contains that individual. A contradiction is also signalled if the new individual is an element of the known non-members of the instance.

2. ADD-NON-ELEMENT adds a new individual to the list of known non-members unless the individual is known to be a member.

3. ASSIGN-TO-CONSTANT makes the list inside an instance equal to a given list and marks the instance closed. If the instance already contains elements, they must all be members of the constant set or a contradiction is signalled. Also, no elements of the constant set can be known non-members.

4. EQUAL? takes another set instance as an argument and returns *true* if both instances are closed sets and have the same members; returns *false* if they are closed and do not contain the same members or if one set has known non-members that are members of the other; otherwise it returns *unknown*.

Set also answers the EQUATE message. When an instance of **set** receives a message of the form

EQUATE(other-object),

it responds with a single instance that contains the union of the original instance's members and the members of other-object. The new instance's list of non-members is the union of the non-members of the originals. An EQUATE will signal a contradiction unless one of the following is true:

1. both original instances are closed and their lists of elements are equal

2. exactly one of the original instances is closed, the elements of the open instance are also members of the closed instance, and none of the known non-elements of the open instance are members of the closed instance

3. both instances are open, the non-members of one are disjoint from the members of the other, and vice versa.

7.3 Solving the Example Problem

This section explains how the small FAMILIES problem is solved with the representation designed for it. The initial problem statement is shown in figure 7.1. The final problem statement is shown in figure 7.2. The boxes in the figure enclose statements captured by the specialized representation and the statements marked by (*) are captured by operationalization. The statement marked by (**) is captured in the in type hierarchy of representations. The final representation is shown in figure 7.3.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child\text{-}of(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all $x \mid parent(S, x)$

Figure 7.1: The small FAMILIES problem.

The system creates a problem situation from the three specific statements found in the final formulation of the problem. Let us illustrate how it does this beginning with the statement

$$children'(parent\text{-}set\text{-}of(P)) = \{R\}.$$

To create a situation representing this statement, an instance `P` of `family-member` is created; an instance `parent-set-1` of `parent-set` is created; and `P` is added to it, then the `parent-set-of(P)` is made `parent-set-1`. The system has, so far, created a situation representing $parent\text{-}set\text{-}of(P)$. Next, a child-set is created, call it `child-set-1`, assigned the constant value $\{R\}$, and this object is made the `children` of `parent-set-1`. A diagrammatic version of the structure built for this statement is shown in figure 7.4.

Next the system creates a representation of the statement

$$couple(Q) = couple(P) \wedge Q \neq P$$

$children'(parent\text{-}set\text{-}of(P)) = \{R\}$
$couple(Q) = couple(P) \wedge Q \neq P$
$\exists z[children'(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children'(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$

$\forall x \forall y[child\text{-}of(x,y) \Leftrightarrow parent(y,x)]$
$(**)\forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$(*)\ \forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$(*)\forall x\ x \in parent\text{-}set\text{-}of(x)$
$(*)\forall x\ x \in couple(x)$

$\forall x \neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$
$\forall x\ child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(x))$
$\forall x \forall y[\ child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \Rightarrow child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z[(child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \wedge child\text{-}set(z) = children'(parent\text{-}set\text{-}of(y))) \Rightarrow child\text{-}set(z) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z \forall w[\ child\text{-}set(x) = children'(parent\text{-}set\text{-}of(y)) \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(z)) \wedge y \neq z \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(w)) \Rightarrow w = y \vee w = z]$

Query: find-the $parents'(child\text{-}set\text{-}of(S))$

Figure 7.2: Final formulation of the small FAMILIES problem.

```
(deftype family-member specializes unique-individual disjoint)
couple:  function(family-member,married-couple)
(deftype married-couple specializes fixed-size-disjoint-set(2,family))
(deftype child-set specializes disjoint-set(family-member))
child-set-of:  function(family-member,child-set)
(deftype parent-set specializes
fixed-size-disjoint-set(2,family-member))
parent-set-of:  partial-function(family-member,parent-set)
children':  1-1-partial-function(parent-set,child-set)
parents':  1-1-function(child-set,parent-set)
```

Figure 7.3: Final representation of the FAMILIES problem.

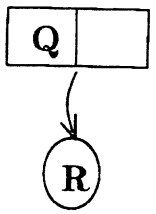


Figure 7.4: Diagram of representation for $children'(parent\text{-}set\text{-}of(P)) = \{R\}$.

and adds it to the problem situation. To accomplish this, it creates an object `Q` and a couple containing `Q`. It creates another couple and adds the object `P` to it. It EQUATES the two couples just created producing a single couple containing `P` and `Q`. Note that since `Q` and `P` are instances of **`unique-individual`**, the second conjunct is already represented.

Because $couple(Q) = couple(P)$, the procedure that operationalization generates to capture the constraint between couples and parent sets now makes $parent\text{-}set\text{-}of(P) = parent\text{-}set\text{-}of(Q)$. This causes a single **`parent-set`** to be created containing P and Q. This situation is diagrammed in figure 7.5. Note that in the diagrams, the box with two slots is overloaded: it is used to represent both couples and parent sets.

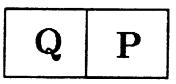


Figure 7.5: Diagram of representation for $couple(Q) = couple(P)$.

The final step in creating the problem situation is to add the representation of the statement

$$\exists z[\ children(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge \\ children(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)].$$

The representation of this statement alone is shown in figure 7.6. The existential variable z is represented in the actual structure with an anonymous instance of **`family-member`**, call that instance **`z`**. Since $children'$ is one-to-one, as the system creates the structure representing this statement, it enforces the one-to-one constraint by equating the objects denoted by $child\text{-}set\text{-}of(z)$ with the $child\text{-}set\text{-}of(R)$. Since an individual is always a member of his own child set and since $child\text{-}set(R) = \{R\}$,

the anonymous individual **z** is equated with **R**. This results in the structure shown in Figure 7.7.

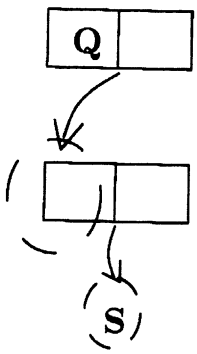


Figure 7.6: Diagram of representation for $\exists z[children(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$.

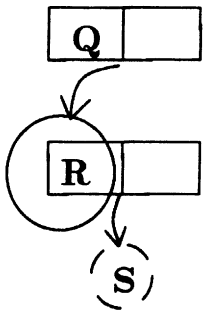


Figure 7.7: Diagram of representation for all three statements.

The system now answers the problem query by inspecting the parents of S which is a *parent-set*. The system knows that this set has two individuals but that it only knows one of them, R. So it answers the question with R and an indication that there are other members that it does not know.

Chapter 8

Related Work

This section highlights the differences between my research and previous work in the following areas:

- Solving word problems
- Automatic programming
- Research on good representations
- Problem reformulation
- Mental models

8.1 Solving Word Problems

Since my representation design system solves word problems, one might expect there to be an interesting relationship between it and previous systems that do this. This section discusses the relationship between my research and two previous efforts to solve word problems and concludes that the relationship is only superficial: All of the systems solve word problems, but the underlying research objectives are very different.

In [Bobrow68], the author reports on a program called STUDENT that solves high school algebra word problems. Bobrow was interested in issues of translating problems

stated in natural language to simple algebraic equations. He was also interested in a psychological model of high school student's performance in this activity and used his model to make predictions about human performance.

The most important difference between STUDENT and my work is that the objective of STUDENT was to translate a problem, stated in English, into a single pre-established target representation. My system is concerned with designing good target representations. For example, in most cases, confronted with a problem like those that STUDENT solved, I would expect my system to select algebraic equations as the target representation.[1] For high school algebra word problems, there would not be much representation design to do. However, given a different kind of problem, my system should (and does) design different kinds of representations.

Another important difference between Bobrow's and my work is that he was interested in the psychological implications of his model. I have not concentrated on this in my work. Like many other efforts in AI the methods my system employs are inspired by human performance, but my emphasis has been on designing good representations regardless of whether or not I have captured the design process that people employ.

The other word problem system I compare my system to is the work reported in [Novak76]. This system solved physics word problems in the area of rigid body statics. Like Bobrow's, this work was concerned with translating a problem, stated in English, into a single target representation and then solving it. However, Novak points out that physics problems of this type are not deductive. The hard part of solving them is figuring out what assumptions to make so that the problem decomposes into idealized pieces to which physical principles can be applied directly. By contrast, the problems that my system works on are deductive and do not require making sophisticated assumptions to simplify the situation presented in the problem.

Another important difference between both of the research efforts discussed here and mine is that they considered the natural language translation problem an important part of what their systems did. I have not considered natural language translation

[1] I have not tried my system on problems like those that STUDENT solves. However, my system does design representations in terms of equations. For example, part of the representation that is designed for the FAMILIES problem is a collection of equations relating sets of children, siblings, brothers, and sisters.

in my research because I believe that analytical reasoning problems are stated in a restricted enough subset of English that the translation problem can be solved by "off the shelf" natural language technology.

8.2 Relationship to Automatic Programming

Since my system generates programs, it can be viewed as an automatic programming system. This section argues that the major difference between my work and other automatic programming systems is that they perform tasks such as algorithm design, data structure selection, and optimization (of algorithms and data structures) within a fixed representation which is chosen by a person prior to the point where the automatic programming system gets involved. By contrast my system is concerned with those earlier steps in the problem solving process during which a representation is designed. There are a number of ways to demonstrate this distinction. One way is to compare the input to automatic programming systems with the input to my system.

Automatic programming systems begin with a specification of a program as input. In contrast, my system starts with a specification of a problem. For instance, the SAFE system is an automatic programming system that accepts an informal specification of a program and formalizes it. Figure 8.1 is an example of a specification given to SAFE. This specification is process oriented. Analytical reasoning problem specifications clearly are not.

I chose SAFE as one effort to discuss because, unlike most work in automatic programming, it is concerned with completing informal specifications. Like my system, SAFE tries to acquire missing information. The techniques it uses identify incompleteness in a natural language specification based on syntactic cues in the text. In contrast, my system relies on the semantic properties of library structures to guide a search for missing information.

Another point of distinction between the two systems has to do with irrelevance. Informal program specifications do not appear to contain irrelevant information and SAFE does not consider this possibility at all.

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER)))

(IF ((ONE)
     (IS FOUND))
    ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
     (IS COMPUTED) ACCORDING TO ("FORMULA-1"))

((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING TO ("FORMULA-2"))

(WHEN ((THE TRANSMISSION TIME)
       (HAS BEEN COMPUTED))
      ((IT)
       (IS INSERTED)
       AS (THE (PRIMARY ENTRY))
       IN (A (TRANSMISSION SCHEDULE))))

(FOR (EACH PATS ENTRY)
     (PERFORM)
     (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
         (IS COMPUTED) ACCORDING TO ("FORMULA-1"))
        ((THE RATS'S (CLOCK TRANSMISSION TIME))
         (IS COMPUTED) ACCORDING TO ("FORMULA-2")))))

((THE RATS (TRANSMISSION TIMES))
 (ARE ENTERED)
 INTO (THE SCHEDULE))
```

Figure 8.1: Example of a specification given to SAFE.

The distinction I have drawn between specifying programs and problems seems clear enough in the example just given. However, it becomes less clear for systems such as [Cohen86] whose input is a predicate calculus specification of a set to generate or a condition to test. Figure 8.2 gives an example of the input to Cohen's system which is called AP5. This specification describes more of what a programmer wants the machine to do than how it is to do it.

```
(DeclareRel Sex 2) ; a binary relation
; e.g., (Sex Sam Male)
(DeclareRel Parent 2) ; (Parent child parent)
(DefineRel Sibling (x y)
   (and (not (eq x y))
        (Exists (parent)
                (and (Parent x parent)
                     (Parent y parent)))))
(Defun list-nephews (person)
  (loop for nephew s.t.
    (and (Sex nephew 'male)
         (Exists (sibling)
           (and (Sibling person sibling)
                (Parent nephew sibling))))
    collect nephew))
```

Figure 8.2: A specification input to AP5.

Note that we could easily define a similar analytical reasoning problem that provides a definition of the *sibling* and *nephew* relation and then asks the question

find-all $x \mid nephew(P, x)$.

However, AP5 and my system address very different problems. AP5 is given a specification and a collection of annotations that select representations for the primitive relations in the specification. It compiles the specification given information it has about the costs of different ways of testing relations and for generating n-tuples of individuals standing in a relation. Once the person observes the behavior of the program that AP5 generates, he/she can choose better representations for the primitive relations and use AP5 to recompile the specification. The important point to notice is that the person selects the representations and AP5 produces the best code it can

based on those selection. My system chooses the representations.

I once conceived of my system producing a specification instead of a program. I had in mind that this specification would then be handed to an automatic programming system such as [Barstow79] (or possibly AP5) which would then make data structure selection and algorithm optimization decisions. I was subsequently persuaded to produce a program instead. Still, the code generation part of my system does not try to perform data structure selection or algorithm optimization. For example, my system is concerned with deciding that some problem concept is best represented as a set. When a program is generated, the representation design system uses a default implementation for sets instead of trying to choose from among alternative implementations (e.g., a list or a bit vector) as, for example, [Rovner76] does.

More conventional automatic programming systems such as QA3 (described in [Green68]) take the approach that automatic programming is a theorem proving activity in which the system tries to prove the existence of some entity that we want a program to produce. A human provides the system with a theory of the operations available for writing programs and when the system uses these to prove a theorem, it produces the desired program as a by-product. For example, to generate a program that sorts a list, QA3 tries to prove a theorem stating that for all (finite) lists there exists a sorted version. Since it has been provided with axioms that describe list operations and operations for comparing numbers, a by-product of the proof is a program that sorts lists.

As Green and others have pointed out, the formulation of the set of axioms that describe both the operations available for programming and the desired program to be written can have a dramatic effect on whether the theorem proving approach succeeds. My work has concentrated on how better formulations of a set of axioms can be found automatically.

Operationalization is the activity that my system engages in that is most similar to conventional automatic programming in that it transforms predicate calculus statements into code fragments. It is a restricted form of automatic programming that is used as a last resort in capturing problem constraints. It can not produce recursive programs and can only produce simple forms of iteration.

8.3 Research on Good Representation

As I have already stated, the principle difference between my research and previous efforts to understand what makes a good representation is that those efforts were concerned with recognizing the properties of good representations, while my research is about generating such representations prospectively. Still much of this work provided me with a good starting point and also helped me to sort out what the important issues about representation are. Several works give good explanations of what it means for a representation to be direct (or analogical) and what the consequences of having such a representation are (see [Sloman71, Sloman85, Hayes74, Lenat & Brown 84]).

Pylyshyn's work, reported in [Pylyshyn75], helped me to understand a phenomenon that I observed in the representations that people design to solve analytical reasoning problems. Specifically, it does not make sense to talk about the directness of the structure of a representation devoid of the interpretation functions that give a semantics to that structure. For example, even in an obvious case like using a structure with two slots to represent married couples, it is really the interpretation provided by the procedures that manipulate couples that preserve the relationship between the representation and couples in the "real" world.

8.4 Problem Reformulation

Some early work on problem reformulation can be found in [McCarthy64, Newell65], and [Newell66]. Instead of trying to exhaustively compare my work with these, section concentrates on three works in the area. One work reported in [Amarel68] is included because of its influence in the area. The two other works [Korf80, Subramanian87] are included because they illustrate recent work in the area. Special attention is paid to [Korf80] because the work reported there is the most direct ancestor of mine.

Amarel's work is a paper and pencil study leading us through a successive refinement of representations for the familiar Missionaries and Cannibals (M&C) problem. The work differs from mine in two ways. First, it considers a substantially different domain: reasoning about the effects of actions. Second, it does not seriously consider issues in automating the search for a refinement sequence. The resulting characteriza-

tion of the transformations used and space of possible representations being searched is loose. In contrast, these issues have been at the forefront of my concerns.

Amarel leads us through a sequence of refinements and, at each step, identifies some interesting properties of the M&C problem. Then he discusses a refined representation that capitalizes on those properties. Many of the transformation discussed are informally motivated by arguments about the search space generated in finding solutions. For example, in the first version of the problem, the operators encode the possible moves and the non-cannibalization conditions are expressed as general constraints on the problem. In the next version, the general constraints are "compiled" into the operators so that the new operators are applicable only when they produce a non-cannibalized (legal) state. The resultant representation (with constraints compiled into the operators) is better because the search space is smaller.

In a number of places in his argument, Amarel fails to define what is involved in the transformations he offers. For example, he begins with representations in terms of production systems in which rules construct new states from old ones. At one point he switches to a reduction system. Reduction systems begin with a problem stated as

initial state $\Rightarrow$ *final state.*

This is interpreted as "the final state is attainable from the initial state." The process of solving the problem involves "reducing" this statement to a sequence of states attainable by application of primitive operators. The problem with this reformulation (from production systems to reduction systems) is that we are not told how such a switch is made or when it is advantageous to do so. In fact, it turns out that this transformation is not advantageous for M&C.

In another part of the paper, problem representations are described in terms of state space graphs. At one point, Amarel structures these graphs by viewing them as juxtaposed two dimensional grids. Unfortunately, the reader is left with little feeling as to what the precise nature of this transformation is and the conditions under which such a transformation is useful. In this presentation he exploits our visual abilities to notice certain properties. But what are these properties? And how did Amarel identify them in the problem?

Korf's work, reported in [Korf80], has a similar mind set to that of Amarel's. However, Korf went much further to develop a formalism for describing representations and transformations between different representations. He also provided a characterization of two dimensions along which different transformations effect representations: viewing them as homomorphic and isomorphic transformations in a space of possible representations. Homomorphic transformations preserve structure and reduce information content. Isomorphic transformations preserve information content but change a representation's structure.

An important contribution of his work is that Korf demonstrated how representation changes that had been previously viewed as "leaps of insight" could, in fact, be modeled as gradual refinement involving transformations of the type he identifies.

One of his examples is the mutilated checker board. He describes transformations along the way to a representation consisting of two integers, one representing the number of uncovered black squares and one representing the number of uncovered red squares. One mapping involves assuming that the squares are indistinguishable. Then any situation in which the same number of squares are uncovered can be thought of as the same. This can be modeled as a homomorphic transformation that maps board situations into sets of indistinguished uncovered squares. Since all that is important about the sets of squares is their cardinality, the set representation can be transformed into one in which the sets are replaced by integers representing their cardinality. This transformation can be modeled as an isomorphic mapping between sets and integers representing their cardinality.

In some ways my work can be seen as an extension of Korf's. He was concerned with characterizing a space of possible representations and types of transformations on them. My work is concerned with *how to choose the right transformations to do to arrive at a good problem representation.* I have identified some of the essential properties of representations and given a method to design representations with those properties.

Korf (and Amarel) viewed problem solving as state space search and observed that changes in representation (i.e., the description of a problem state) affect the size of the space. The focus of my work has been explaining *how* representations do this

and how to design representations that yield smaller search spaces. The claim is that when a representation captures more constraints in its structure and behavior the problem solving space is reduced.

One point that Korf does not make clear is that he actually describes two different kinds of homomorphic transformation: transformations on search spaces and transformations on state descriptions. An example of a transformation on a search space is introducing uninterruptible operator sequences (macro-operators, lemmas, etc.) and then removing the original operators. This has the effect of reducing a search space by "skipping over" intermediate states derived by the original operators. An example of a transformation on a state description is collapsing a checkerboard into two integers representing the cardinality of the set of red squares and black squares respectively. This has the effect of reducing a search space by throwing away distinctions in a state description and thereby grouping states into equivalence classes. These two different kinds of homomorphisms have the same effect on a search space: the overall size of the space is reduced. However, the reasoning involved in performing them is very different.

My research has only considered one method of collapsing state descriptions in its use of the irrelevance filter (described in Chapter 3) which can be viewed as removing information from a state description that is irrelevant to solving a problem. One can imagine other homomorphic transformations (such as those suggested in Korf's further work section) whose effect is to remove information without changing a problem's solution.

My research does consider homomorphic transformations on search spaces because it identifies specialized structures for representing. These structure have specialized procedures associated with them. Such procedures enforce consequences directly in a representation, skipping over intermediate steps necessary for a theorem prover to deduce those consequences.

[Subramanian87] reports on a study in which logic is used as a tool to investigate properties of irrelevance. The paper discusses the application of their theory to three example problems including proving that a particular reformulation is justified because it removes only information that is irrelevant to solving a problem. This

example applies the theory to justify the removal of intermediate links in an ancestral tree for a problem that asks whether two individuals are in the same family. The intermediate ancestral links are irrelevant because all that matters to answering the question is the relationship between a person and the root of his/her family tree.

In some ways this work is similar to my work on the irrelevance filter. There are two differences. First, my procedure for identifying and removing irrelevance is computationally tractable. Second, the modifications made to a problem in Subramanian's work are more subtle, e.g., viewing the removal of irrelevance as reformulation. The problem given in the paper is initially stated in terms of *father*, *ancestor*, and *samefamily* and is reformulated in terms of *foundingfather* and *samefamily*. This example provides one instance of a kind of reformulation different from the kind I have studied. My reformulations are guaranteed not to change the semantics of a problem. In contrast, the example given in this paper is a reformulation that changes the semantics of the problem while preserving the correctness of the solution. It would be interesting to pursue this type of reformulation in automatic representation design.

8.5 Psychology and Mental Models

There is a body of work in psychology on human mental models. The idea in this theory is that people solve problems by building and examining concrete integrated structures that are based on perceiving or imagining the events (situations) described in text[Johnson-Laird82]. The way these structures appear to be built and examined has some intriguing similarities to our specialized representations.

Chapter 9

Summary and Future Work

9.1 Summary

The contribution of this thesis is a technique that, given a problem, designs a representation specialized to a small class containing that problem. The representations designed are specialized because they capture the constraints of a problem class in their structure and behavior. When a problem solver uses a specialized representation to solve a problem, it is restricted so that it can express only situations in which the constraints of the class are satisfied. As a result, the space that the problem solver considers is significantly reduced, increasing problem solving efficiency.

An abstract characterization of the design technique is as follows. The input to the process is a problem statement, the representation mapping for the concepts in that statement, and a collection of available structures with axioms describing the kinds of constraints that they can capture. The constraint on a concept is captured when it follows from the axioms of the structure representing it. The system tries to modify the representation mapping (i.e., to select different structures to represent the problem concepts) so that the constraints on those concepts are captured.

The technique is implemented by three processes called *classification*, *concept introduction*, and *operationalization*. Classification and concept introduction run as coroutines to capture as many of the constraints of a problem as possible. Classification specializes the representation of individual concepts (relations, functions, etc.), while concept introduction introduces related concepts, attempting to find better for-

mulations in which classification captures more of the problem's constraints. Usually these processes are unable to capture all of a problem's constraints, so representation design includes operationalization, a "mopping up" process. It captures the constraints left uncaptured by the previous processes by writing new procedures. Such procedures enforce constraints by responding when a constraint is violated, adding new information to a problem situation to reestablish the constraint.

Classification has a library of *structures* which are implemented as parameterized abstract data types. These are used as prototypes for creating representations that enforce constraints through a combination of syntactic structure and behavior (procedures). Representations created from more specialized structures capture more constraints. For example, two of the structures in the hierarchy are **function** and **1-1 function**. A representation created from **1-1 function** is more specialized than one created from **function** because it captures the additional constraint that every range element is the image of at most one domain element.

Classification "pushes" concepts down into a hierarchy of concept classes. Some of these classes have library structures associated with them. The library structure associated with a class is used to create representations for concepts in that class. Such representations capture all of the constraints of the class. For example, the hierarchy contains the class of all symmetric binary relations. This class has the structure **sym-rel** associated with it. Representations created from **sym-rel** capture the symmetry constraint.

Successful classification identifies the most specialized structure (or collection of structure) with which to represent the concept. When a representation is specialized, fewer situations can be expressed in it. For example, when "parents" is represented as a **function** we can express situations in which an individual has more than one pair of parents; when it is represented as a **1-1 function** such situations can not be expressed. Thus specialized representations reduce the space that a problem solver must consider.

Given a problem, classification comes up with a collection of maximally specialized representations for its concepts. However, classification by itself has a serious limitation: Its success depends on the particular vocabulary used to state a problem.

The FAMILIES problem, for example, is stated in terms of *married*, which is classified beginning with **relation**. None of the specializations of a **relation** capture the fact that married couples are all of size two. However, if the problem had been stated in terms of *couples*, classification would have been more successful because a specialization of **set** takes advantage of size constraints.

Concept introduction overcomes the limitation of classification by introducing related concepts when classification does not capture all the constraints of a concept. Concept introduction is implemented by rules that are attached to nodes in the taxonomy of library structures. When a node is reached that has an associated introduction rule, the rule is applied, introducing one or more new concepts. The effect of introducing a new concept is to represent it differently, giving classification access to different parts of the taxonomy: different library structures capture different constraints and have different specializations. Representing a concept differently often allows classification to find a better fit between the constraints on that concept and the constraints captured by library structures.

Introducing new concepts also reformulates a problem. This is accomplished by treating the logical definition of a new concept as a rewrite to perform on problem statements. Reformulation is useful for two reasons. First, it often allows the system to recognize new properties of a concept, facilitating further classification. Second, operationalization is often able to capture reformulated statements with more efficient procedures.

As new concepts are introduced, the representation design system explores a space of alternative problem formulations. For example, introducing *couple* for *married* creates two alternative formulations: one in terms of *married* and one in terms of *couple*. Alternative problem formulations are maintained because the representation design system can not tell whether an introduced concept will capture more constraints than an existing one until the new concept is fully classified.

Classification extended by concept introduction is called *extended classification*. What is interesting about extended classification is that the two processes that are involved in it are fairly simple, however, the behavior of the combination of classification and introduction can result in sequences of reformulations that change a problem

significantly. For example, consider the introductions that result from extended classification of *married*:

1. The concept *spouses* is introduced. This is a function from individuals to the sets of individuals to whom they are married.

2. The concept *non-empty-spouses* is introduced. This is a partial function from individuals to the non-empty sets of individuals to whom they are married.

3. The concept *spouse* is introduced. This is a partial function that captures the fact that individuals have at most one spouse.

4. The concept *couple* is introduced. This is a partial function from individuals to the married couple that they are members of. `Couple` captures the following facts: not all individuals are married, each married couple is disjoint from all other married couples, married couples contain exactly two members.

Operationalization tries to capture the constraints of any statements remaining after extended classification by writing new procedures and using these to specialize the representations created by classification and concept introduction. For example, suppose the statement

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

is left uncaptured and that *siblings* is represented as a function from individuals to their set of siblings (i.e., the range elements of *siblings* are represented as **sets**). Operationalization captures the constraint of this statement by writing procedures that watch for the addition of facts violating the statement's constraint. One such fact has the form $x_1 \in \mathit{siblings}(y_1)$. Accordingly, one procedure that operationalization writes watches for the addition of facts of this form and responds by adding a fact of the form $y_1 \in \mathit{siblings}(x_1)$, reestablishing the constraint.

When operationalization succeeds in writing a procedure like the one above for every fact whose addition to a problem situation can violate a statement's constraint, the statement is captured.

When all of the constraints of a problem class are captured in a representation, it is used to solve the problem as follows. First, the specific statements of the problem

are expressed in the specialized representation. As this is done, a data structure is created that represents the situation described in the specific problem. For example, Q is married to P in the small FAMILIES problem. In the specialized representation designed for this problem, this is expressed as, "Q and P are in the same couple." When the relationship between Q and P is expressed in this way, an instance of `couple`, containing Q and P, is added to the problem situation.

The data structure created when the problem specifics are expressed in a specialized representation is then inspected for the problem solution. For example, the question in the small FAMILIES problem is, "Who are the parents of S?" This question is answered by accessing the `parents` of `S` in the data structure created for the problem.

The representation design system has been tested on eight analytical reasoning problems. It successfully captures all the constraints of these problems. However, in general, representation design can fail to capture some constraints of a problem. In this case, the result of representation design is a specialized representation and a *smaller* collection of statements (the uncaptured ones) that the problem solver must reason about explicitly in that representation. In effect, the problem solver uses the specialized representation to accelerate the problem solving process in the same way that specialized reasoners have been used to accelerate theorem proving.

9.1.1 Summary of An Example of Representation Design

This section summarizes the design of the specialized representation for the small FAMILIES problem used as an example throughout this thesis. The problem is shown again in Figure 9.1.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$married(Q, P)$
Query: find-all $x \mid parent(S, x)$

Figure 9.1: The small FAMILIES problem.

Before representation design begins, the system tries to acquire definitions missing

from the problem statement, attempts to eliminate irrelevant information, and develops a description of the initial representation. These activities are detailed in chapter 3.

For this problem, the representation design system begins by prompting the user for definitions of the concepts mentioned. The user supplies definitions for *grandchild* and *parent*:

$\forall x \forall y [grandchild(x, y) \Leftrightarrow \exists z (child(x, z) \wedge child(z, y))]$
$\forall x \forall y [parent(y, x) \Leftrightarrow child(x, y)]$.

Then, the system runs the irrelevance filter and finds that *married* is irrelevant. Before eliminating it, the system asks whether there are necessary or sufficient conditions for *married* in terms of other concepts mentioned. The user responds with both:

$\forall x \forall y \forall c [child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow married(x, y)]$
$\forall x \forall y \forall c [married(x, y) \wedge child(x, c) \Rightarrow child(y, c)]$.

With these statements added, *married* becomes relevant.

Next, the system describes the problem's initial representation. In general, such descriptions include definitions for three types of concepts: the primitive concepts, concepts that have relevant mixed constraints on them, and concepts that appear in find-all queries. The description derived for the example problem is:

```
(deftype family-member specializes unique-individual disjoint)
married:  relation(family-member,family-member)
child:  relation(family-member,family-member)
parent:  relation(family-member,family-member)
```

Parent is included because it appears in a find-all query. Including it allows the system to consider the cost of answering the find-all query by considering the alternative formulations generated during extended classification of *parent*.

Next, the system performs extended classification on the concepts whose representations were defined above. We begin (arbitrarily) with *married*. Extended classification of *married* results in a sequence of introductions yielding the concept *couple*; details of this were given in section 5.5.

As a result, a representation is defined for *couple* as

```
couple: function(family-member,married-couple)
```

and the sort *married-couple* is defined as

```
(deftype married-couple specializes
fixed-size-disjoint-set(2,family-member)).
```

The problem statement is also reformulated and a number of statements are added by knowledge acquisition activities. The new problem statement is shown in Figure 9.2. Note that the statements enclosed in the box in that figure are captured by `couple`.

$P : family\text{-}member$, $Q : family\text{-}member$,
$R : family\text{-}member$, $S : family\text{-}member$
$grandchild(Q, S)$
$\forall x\ child(P, x) \Leftrightarrow x = R$
$couple(Q) = couple(P) \wedge Q \neq P$
$\forall x \forall y[grandchild(x, y) \Leftrightarrow \exists z(child(x, z) \wedge child(z, y))]$
$\forall x \forall y[parent(y, x) \Leftrightarrow child(x, y)]$
$\forall x \forall y \forall c[child(x, c) \wedge child(y, c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge x \neq y]$
$\forall x \forall y \forall c[couple(x) = couple(y) \wedge x \neq y \wedge child(x, c) \Rightarrow child(y, c)]$

$\neg[couple(x) = couple(x) \wedge x \neq x]$
$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$

Query: find-all $x \mid parent(S, x)$

Figure 9.2: Example problem reformulated in terms of *couple*

The system now performs extended classification on *child* which results in the following representations:

```
(deftype child-set specializes disjoint-set(family-member))
child-set-of: function(family-member,child-set)
(deftype parent-set specializes
fixed-size-disjoint-set(2,family-member))
parent-set-of: partial-function(family-member,parent-set)
children': 1-1-partial-function(parent-set,child-set)
parents': 1-1-function(child-set,parent-set)
```

Again, statements get added to the problem and it gets reformulated. The result of this is the problem statement in Figure 9.3. As in the previous figure, the statements enclosed in the box have their constraints captured by the specialized representations. Also note that the two statements marked by (*) were derived by the rewrite system from the two statements

$\forall x \forall y \forall c[child(x,c) \wedge child(y,c) \wedge x \neq y \Rightarrow couple(x) = couple(y) \wedge x \neq y]$
$\forall x \forall y \forall c[couple(x) = couple(y) \wedge x \neq y \wedge child(x,c) \Rightarrow child(y,c)]$

The first of these is recognized as a subsumption constraint between *parent-set* and *couple* and captured in the type hierarchy of representations.

$children'(parent\text{-}set\text{-}of(P)) = \{R\}$
$couple(Q) = couple(P) \wedge Q \neq P$
$\exists z[children'(parent\text{-}set\text{-}of(Q)) = child\text{-}set(z) \wedge children'(parent\text{-}set\text{-}of(z)) = child\text{-}set(S)]$

$\forall x \forall y[parent(x,y) \Leftrightarrow child\text{-}set\text{-}of(x) = children'(parent\text{-}set(y))]$
$(*) \forall x \forall y[parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y) \Rightarrow couple(x) = couple(y)]$
$(*)$ $\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]$
$\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)]$
$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)]$
$\forall x\ x \in child\text{-}set\text{-}of(x)$

$\forall x \forall y[couple(x) = couple(y) \wedge x \neq y \Leftrightarrow couple(y) = couple(x) \wedge x \neq y]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(y) = couple(z) \wedge y \neq z \Rightarrow couple(x) \neq couple(z) \vee x = z]$
$\forall x \forall y \forall z[couple(x) = couple(y) \wedge x \neq y \wedge couple(x) = couple(z) \wedge x \neq z \Rightarrow y = z]$
$\forall x\ child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(x))$
$\forall x \forall y[\ child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \Rightarrow child\text{-}set(x) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z[(child\text{-}set(y) = children'(parent\text{-}set\text{-}of(x)) \wedge child\text{-}set(z) = children'(parent\text{-}set\text{-}of(y))) \Rightarrow child\text{-}set(z) \neq children'(parent\text{-}set\text{-}of(y))]$

$\forall x \forall y \forall z \forall w[\ child\text{-}set(x) = children'(parent\text{-}set\text{-}of(y)) \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(z)) \wedge y \neq z \wedge child\text{-}set(x) = children'(parent\text{-}set\text{-}of(w)) \Rightarrow w = y \vee w = z]$

Query: find-all $x \mid parent(S,x)$

Figure 9.3: Formulation of example problem after *child* has been classified.

Before proceeding with the extended classification of *parent*, the system attempts to create a find-the query from it that accesses an existing set, i.e., it transforms the find-all query into

find-the $\{x \mid parent(S,x)\}$.

This is expanded using the definition of *parent* into

find-the $\{x \mid child\text{-}set\text{-}of(S) = children'(parent\text{-}set\text{-}of(x))\}$.

The rewriting system transforms this into

find-the $parents'(child\text{-}set\text{-}of(S))$.

Since it has succeeded in deriving a find-the query that accesses an existing representation (a *parent-set*), the system expands any other occurrences of *parent* using its definition, removes its definition, and does not perform extended classification on it.

If *parent-set* had not be created during the extended classification of *child*, the system would not have been able to simplify the original find-all query as shown. In this case, the system would proceed by performing extended classification of *parent*, introducing a concept for sets of parents and comparing the new formulation of the problem with the formulation in terms of *parent*.

Extended classification now terminates and operationalization tries to capture any constraints left over. The uncaptured statements are:

$$\forall x\ x \in child\text{-}set\text{-}of(x) \tag{1}$$
$$\forall x[couple(x) \neq \perp \Rightarrow x \in couple(x)] \tag{2}$$
$$\forall x[parent\text{-}set\text{-}of(x) \neq \perp \Rightarrow x \in parent\text{-}set\text{-}of(x)] \tag{3}$$
$$\forall x \forall y[parent\text{-}set\text{-}of(x) \neq \perp \ \wedge couple(x) = couple(y) \Rightarrow parent\text{-}set\text{-}of(x) = parent\text{-}set\text{-}of(y)]. \tag{4}$$

(1) is compiled into a daemon that responds to the creation of a range element, $child\text{-}set\text{-}of(x)$, by adding x to that child set; (2) and (3) are compiled into similar procedures. The procedure generated for (4) does the following. When a parent set is created for an individual x his/her couple is made the same as the couple of the other individual in the parent set.

9.2 Future Work

9.2.1 Experimental Work

The current system knowledge bases are the result of an analysis of twenty representative analytical reasoning problems (for a discussion of where these came from, see

Chapter 1). To date, the system has only been tried on three of these (and several variations of each, twelve problems in total). The most immediate continuation of the experimental work would be to try the other seventeen problems (and variations of them). Chapter 1 argues that this effort should not require any modification to the representation design algorithm and only small additions to the knowledge bases.

Another avenue of experimentation is to study deductive problems other than analytical reasoning problems found on GREs. For example, I have begun studying Schubert's Steamroller. This problem is interesting because it is combinatorially difficult enough that when it was first proposed (1978) no automated theorem provers were able to solve it. The problem has now been solved by a number of theorem provers and there have been discussions in the automated reasoning literature about different theorem proving strategies and formulations of the problem (see [Stickel86]).

The systems that solve Schubert's Steamroller most efficiently are those based on sorted logics. Reformulation is required to get from the initial problem statement to an appropriate sorted formulation. It appears that my system performs many of the steps that people carry out manually to construct a sorted formulation of a problem. Of particular interest is the fact that when I hand simulated the representation design algorithm on Schubert's Steamroller, it found that information required to solve the problem was missing. These facts appear to have been assumed by the researchers who developed the sorted formulations manually.

Another avenue for experimentation is to try the representation design system on some of the problems appearing over the years in the literature on problem reformulation, e.g., the Missionaries & Cannibals (M& C) problem or the mutilated checkerboard problem. The system requires extension to work on problems that are solved by state space search: In its current form, it can only design representations for the individual states because it designs representations for problems that have a single state of affairs, whose deductive consequences are difficult to compute and are required in answering the questions posed. The design of a good representation of the individual states in the M& C problem turns out to be only a small part of the design effort. The most leverage is gained in this problem by exploiting properties of its search space. To apply my approach here requires my system be given (or to automatically generate) a representation of the initial search space, i.e., the space

induced by the initial representation of state and of the operators.

9.2.2 Extending the Current System's Functionality

There are a number of ways to extend the functionality of the current implementation. One way is to extend the query procedures. The current procedure for determining if a fact is necessarily true in a problem situation simply looks for that fact in the situation. Since the representations that the system designs are incomplete, a fact can follow from a situation but not be found by the above technique. Furthermore, unlike in a complete deductive system, such a fact can sometimes be proved by establishing that its negation is impossible.[1]

We can extend the system's current query procedures so that they are "more complete." The extended procedure for necessity does the following. It looks to see if the fact is represented in a situation. If so, it answers that the fact follows. Otherwise, it adds the negation of the fact to the situation. If a contradiction results, it also answers that the fact follows. If both of these methods fail, it reports that the truth value of the fact is unknown. If desired, it can be further extended to answer that a fact *does not* follow if its negation is present or adding the fact causes a contradiction.

The other query procedures can be extended similarly. Note, however, that while the extended query procedures will be able to answer some questions that the originals could not, they will not alter the fact that a representation designed by the system may be incomplete and, in this case, the query procedures will not be complete.

Another straightforward extension is to design a better evaluation function for estimating the costs of problem formulations. As noted in Chapter 5, the technique for estimating the cost of a statement determines the number of loops in the procedure generated by operationalization to capture the constraint of the statement. It produces an estimate of 1 if the procedure has no loops, n if is has one loop, n^2 if it has a loop nested inside another, and so on. This is a fairly gross estimate of complexity which could be refined in a number of ways. For example, the procedure could produce a more general polynomial that had more than just a most significant term and also had estimated coefficients for the terms. Consider, for instance, the

[1] In a complete deductive system, $-\Diamond\neg\phi$ is equivalent to $\Box\phi$.

statement

$$\forall x \forall y \forall z [R(x,y) \wedge R(y,z) \Rightarrow R(x,z)]$$

for which the current estimate will be n because one loop is required to find bindings for z each time a new pair $< x, y >$ is related by R. The current procedure will give the same estimate for a similar statement that has a conjunction of two atomic formulas in the consequent. However, a more accurate estimate for the latter statement would be $2n$ because two facts are added to a problem situation for each binding of z.

These and other factors could be taken into account by the estimator producing more accurate estimates and this might allow the system to make more refined decisions in comparing alternative formulations.

Another extension is to add more sophisticated techniques for establishing the properties of problem concepts during classification. This is especially important if one tries to extend the system into other domains. For example, section 9.2.1 pointed out that the current version of the system can not design representations that exploit properties of a state space. One reason for this is that more sophisticated techniques are required to establish properties of such a space. For instance, one very useful technique is to begin exploring a state space, look for suggested properties in a portion of the space, prove by induction over operator sequences that the property holds for the entire space, and then use that property to reformulate the problem before continuing to solve it.

This last example suggests another important area of future work: integrating representation design more with the solution process. It appears that people refine their representations as they solve problems. Therefore, it seems likely that the system's performance can approximate a human's only if it can do the same.

9.2.3 Representing Disjunction

In general, the system can not design representations for disjunctive problem situations. The difficulty is best illustrated by an example. Consider the following propositional problem:

$W \vee T$
$R \vee S$
$(W \wedge R) \Rightarrow Q$
$(W \wedge S) \Rightarrow Q$
$(T \wedge R) \Rightarrow Q$
$(T \wedge S) \Rightarrow Q$
Query: $\Box Q$.

Q follows in this problem. However, the representation that the system designs will not determine this alone. The difficulty is that there is no way to create a problem situation representing $W \vee T$ without committing to one of them actually being true. In this problem, the truth value of Q depends only on $W \vee T$ and $R \vee S$ being true.

One way to handle disjunction like this is to use the representation that the system designs to reason by cases. For example, a problem solver could use the above representation by assuming $W \wedge R$ and noting that Q follows; then assuming $W \wedge S$ and noting the same; and so on. By trying all possibilities, it could establish that Q follows. One reason that the system does not design representations that reason by cases to enforce constraint is that doing so is exponential.

Even though the system does not design representations for arbitrary disjunctive situations, it can design representations of certain restricted forms of disjunction. For example, one form of disjunction that it can represent has to do with problems assigning a fixed set of objects to a set of slots in which each object can be in no more than a fixed number of slots. In this case, the system designs a representation for the set of possible objects assigned to each slot. In an initial problem situation, the possibility set for each slot contains all the possible values it can take on. The problem solving process is one of using the problem constraints to remove objects from these possibility sets.

Figure 1.5 shows the PROFESSORS problem, a problem of this type. The representation that the system designs for this problem is a sequence of ten offices. For each office the system represents the set of possible professors in that office. Initially, the possibility set for each office contains all the professors. Problem constraints are used to remove professors from these sets. For example, given the statement "Weis is in office 2," the representation removes all other professors from the possibility set for office 2 and removes Weis from all other possibility sets.

An important way to increase the power of the representation design system is to find other special purpose representations of disjunction like the one just described.

Appendix A

The Rewriting System

This appendix describes the statement rewriter used by all of the subprocesses of representation design. Representation design manipulates statements in a problem in a number of ways. The rewriter plays two important roles in these manipulations: as a statement simplifier and as a goal directed derivation mechanism. As a statement simplifier, it replaces terms in a statement with simpler equivalent terms. For example, any statement containing the term $x \in \emptyset$ is rewritten, replacing that term by $false$.

As a derivation mechanism, it supports classification by looking for cues in problem statements and attempting to derive a statement in a form expected by one of the subprocesses of representation design. One place this is used is in support of classification. For example, classification identifies a size constraint on a set by looking for a statement that bounds the set from above. The rewriter supports classification by trying to derive a statement bounding a set from above from statements containing certain equalities involving universally quantified variables. For instance, the system rewrites the statement

$$\forall x \forall y \forall z[y \in spouses(x) \wedge z \in spouses(x) \Rightarrow y = z]$$

as

$$\forall x \forall y \forall z[y \in spouses(x) \Rightarrow spouses(x) \subseteq \{y\}].$$

Classification recognizes this statement as bounding sets of *spouses* from above.

To ensure that the representation design process remains sound, all the rewrite rules have been checked for soundness, i.e., I have shown for each rule that if it rewrites a statement ϕ as ψ, then $\phi \Leftrightarrow \psi$.

I have not attempted to show termination for the existing rule set. However, in practice the rewriter has always terminated and I believe it is possible to demonstrate termination formally.

A.1 Statement Simplification

As a statement simplifier, the rewriter consists of a collection of rules that look for patterns in statements as they are added to a problem description. When a rule's pattern is found in a statement, the rule rewrites it. As a simple example, one rule looks for a statement containing the literal $x \in \emptyset$. Whenever a statement with this literal is added, the rule rewrites it, replacing that literal by $false$.

To understand the effect that simplification can have on a problem, note that statements are added to problem descriptions at three points during representation design: (i) when the initial problem description is given to the system, (ii) when problem statements are reformulated, and (iii) during each intermediate step in operationalization sequences.

The rewriter can simplify statements given initially, but generally, there is not much simplification (of the sort performed by the rewriter) to do at first. It is unlikely, for example, that an initial statement would contain the term $x \in \emptyset$. The simplifier has most of its impact on statements generated by reformulation and operationalization because these processes make uniform changes in problem statements without regard for their meaning. For example, suppose a problem initially contains the statement

$$\forall x \ \neg brother(M, x)$$

and that during design, the system introduces the concept *brothers*, a function mapping an individual to his/her set of brothers. This causes every statement in the problem that mentions *brother* to be reformulated. Hence, the above statement gets reformulated as

$$\forall x \ x \notin brothers(M).$$

A rewrite rule transforms this statement into $brothers(M) = \emptyset$. This rule has simplified the statement because a statement with fewer variables in it is considered to be simpler than a logically equivalent statement containing more variables. There are a number of other rewrite rules that, like the example above, simplify statements by removing variables.

Rules can also use the properties of the representations in a statement in simplifying. For example, one rule rewrites a subset relationship to an equality when it can determine that the two sets involved have the same cardinality. For instance, the subset term in the statement

$$\forall x \forall y \forall c[\{x, y\} \subseteq parents(c) \Rightarrow married(x, y)]$$

is rewritten to an equality because the range of *parents* is *parent-set* which is a collection of sets of size two.

A.1.1 The Current Collection of Simplifying Rewrite Rules

This section summarizes the current collection of simplifying rewrite rules. These are presented at the knowledge level, avoiding the details of the rewrite mechanism. Note that the expressions in the rules below are patterns to match against statements in the logic, not statements in the logic themselves. The existing rules divide roughly into three categories: (i) simplification of literals containing constant sets, (ii) simplification of literals based on more general knowledge about sets, and (iii) simplification to remove variables.

Constant Set Simplification

This is a representative rather than an exhaustive list of rules in this category.

1. Replace $x \in \emptyset$ by $false$.
2. When c and φ are constants, replace $c \in \varphi$ by $true$ if c is an element of φ, and by $false$ otherwise.
3. When P and Q are constants, replace $P \cup Q$ by the union of P and Q.
4. Replace $\varphi_1 = S - \varphi_2$ by $S = \varphi_1 \cup \varphi_2$ when φ_1 and φ_2 are constant sets and it can be shown that $\varphi_2 \subseteq S$.[1]

General Set Simplification

These are all the rules currently in this category.

1. Replace $\{x \mid x \in S\}$ by S.
2. Replace $S_1 \subseteq S_2$ by $S_1 = S_2$, when $card(S_1) = card(S_2)$. The rule can determine the cardinality of a constant set by inspection of the statement, it determines cardinality of other set expressions by accessing the definitions of the representations in the expression.
3. Replace $\{x \mid P \wedge Q\}$ by $\{x \mid P\} \cap \{x \mid Q\}$.
4. Replace $\{x \mid P \vee Q\}$ by $\{x \mid P\} \cup \{x \mid Q\}$.
5. Replace $\{x \mid P \wedge \neg Q\}$ by $\{x \mid P\} - \{x \mid Q\}$.

[1] Note that the system's matcher handles commutative operators and connectives and symmetric relations. Hence, this rule will also match a statement with a term of the form $S - \varphi_2 = \varphi_1$ because $=$ is symmetric.

6. Replace $\{x \mid x = y\}$ by $\{y\}$.

7. Replace $S - \varphi = \emptyset$ by $S \subseteq \varphi$.

Variable Removal

These are all the rules currently in this catergory.

1. Replace $x = x$ by $true$

2. Replace $x \neq x$ by $false$.

3. Replace $x \notin S$ by $S = \emptyset$ when x is universally quantified.

4. Replace $x = F(y) \wedge x = G(y)$ by $F(y) = G(y)$ in a statement S if x does not appear elsewhere in S.

5. When a statement contains the literal $F(x) = y$, where y is a variable, replace it by $true$ and substitute $F(x)$ for all occurrences of y in the rest of the statement[2].

6. Remove the term t from the consequent of a statement of the form

$$(t \wedge \alpha_1 \wedge \cdots \wedge \alpha_n) \Rightarrow (t \wedge \beta_1 \wedge \cdots \wedge \beta_m).$$

A.2 Goal Directed Derivation

The rewriter performs goal directed derivations to attempt to derive statements of a form that the subprocesses of representation design are expecting. For example, as explained in Chapter 4, classification looks for the properties of a problem's concepts by looking for statements of a predetermined form. The rewriter supports classification by attempting to derive statements in a form that classification will recognize as stating properties that it is looking for. In this mode, the rewriter is activated on particular statements, knows what form of statement it is trying to derive, and either succeeds in doing so or returns the statement to its original form.

A rule that begins an attempted derivation executes when the pattern it is looking for is found in a problem statement. In response, it records that a derivation has begun, marks the statement involved, and records the form of statement it is trying to derive. The derivation process proceeds by repeatedly rewriting the marked statement until either the goal form is derived or no more rewrite rules can be applied. Also, rewrite rules can be restricted so that they are applied only to derivations with certain goals.

[2]This rule is generally known as paramodulation.

Here is an example derivation involving the statement

$$\forall x \forall y \forall z [y \in \mathit{spouses}(x) \wedge z \in \mathit{spouses}(x) \Rightarrow y = z]. \quad (1)$$

One rule looks for a statement containing an equality in its consequent, where at least one side of the equality is a universally quantified variable. When the rule finds such a statement, it begins a derivation whose goal is a statement, concluding that a set is bounded from above. The goal in the example is a statement whose consequent is of the form $\mathit{spouses}(x) \subseteq \varphi$, where φ is any constant set.

The general strategy that the system uses in this derivation is to attempt to rewrite the statement so that the consequent is an implication that can be transformed into the desired relationship, i.e, a statement of the form $x \in S \Rightarrow x \in \varphi$ (where φ is a constant set) which is subsequently transformed into $S \subseteq \varphi$.

The system implements this strategy using the following two rules (which are only applied to statements in derivations of this type):

1. When the statement is of the form
$$(x \in S \wedge \alpha_1 \wedge \cdots \wedge \alpha_n) \Rightarrow x = y$$
and x is a universally quantified variable, [3] rewrite the statement as
$$(\alpha_1 \wedge \cdots \wedge \alpha_n) \Rightarrow \{x \mid x \in S\} \subseteq \{x \mid x = y\}.$$

2. When the statement is of the form
$$(x \in S \wedge \alpha_1 \wedge \cdots \wedge \alpha_n) \Rightarrow (x = y_1 \vee \cdots \vee x = y_m)$$
and x is a universally quantified variable, rewrite the statement as
$$(\alpha_1 \wedge \cdots \wedge \alpha_n) \Rightarrow \{x \mid x \in S\} \subseteq \{x \mid x = y_1 \vee \cdots \vee x = y_m\}.$$

Continuing the example, the first rule rewrites (1) as

$$\forall x \forall z [z \in \mathit{spouses}(x) \Rightarrow \{y \mid y \in \mathit{spouses}(x)\} \subseteq \{y \mid y = z\}].$$

This statement is further rewritten by a simplifying rewrite rule described in the last section. First as

$$\forall x \forall z [z \in \mathit{spouses}(x) \Rightarrow \mathit{spouses}(x) \subseteq \{y \mid y = z\}]$$

and then as

$$\forall x \forall z [z \in \mathit{spouses}(x) \Rightarrow \mathit{spouses}(x) \subseteq \{z\}].$$

[3] This pattern will match any statement whose antecedent is a conjunction containing a term matching $x \in S$ because $\wedge$ is commutative.

Bibliography

[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.

[Bell & Machover 77] Bell, J. and Machover, M., A course in Mathematical Logic, North Holland, 1977.

[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.

[Brachman et.al84] Brachman, R.J, Fikes, R.E., and Levesque, H.J., "KRYPTON: A Functional Approach to Knowledge Representation," in Brachman, R.J and Levesque, H.J. (editors), Readings in Knowledge Representation, pp. 411-429, Morgan Kaufmann, 1985.

[Cohen86] Cohen, D., "Automatic Compilation of Logical Specifications into Efficient Programs," AAAI86, pp.20-25, 1986.

[Cohn88] Cohn, A.G., "Many Many Sorted Logics," Workshop on Principles of Hybrid Reasoning, pp.63-78, 1988.

[Green68] Green, C., "Theorem Proving by Resolution as a Basis for Question-Answering Systems," In Michie, D. (editor), Machine Intelligence 4, Edinburgh University Press, 1969.

[Hayes74] Hayes, P.J., "Some Problems and Non-Problems in Representation Theory," in Brachman, R.J and Levesque, H.J. (edi-

tors), Readings in Knowledge Representation, pp. 3-22, Morgan Kaufmann, 1985.

[Johnson-Laird82] Johnson-Laird, P.N., "Ninth Bartlett Memorial Lecture. Thinking as a Skill," Quarterly Journal of Experimental Psychology, 34A, pp. 1-29, 1982.

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

[Korf83] Korf, R.E., "Learning to Solve Problems by Searching for Macro-Operations," CMU-CS-83-138, 1983.

[Kowalski75] Kowalski, R., "A Proof Procedure Using Connection Graphs," Journal of the ACM, 22, No. 4, 1975.

[Lenat & Brown 84] Lenat, D.B. and Brown, J.S., "Why AM and EURISKO Appear to Work," Artificial Intelligence, 23, pp. 269-294, 1984.

[Marr82] Marr, D., Vision, W.H. Freeman and Company, 1982.

[McAllester82] McAllester, D.A, "Reasoning Utility Package User's Manual," AI Memo 667, M.I.T. Artificial Intelligence Laboratory, 1982.

[McCarthy64] McCarthy, J., "A tough nut for proof procedures," A.I. Project Memo 16, Stanford University, 1964.

[Miller & Schubert 88] Miller, S.A. and Schubert, L.K., "Using Specialists to Accelerate General Reasoning," AAAI88, pp. 161-165, 1988.

[Newell65] Newell, A., "Limitations of the current stock of ideas about problem solving," in Electronic Information Handling, Kent, A,. and Taulbee, O., (editors), Spartan Books, 1965.

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

[Novak76] Novak, G., "Computer Understanding of Physics Problems Stated in Natural Language," in American Journal of Computational Linguistics, Microfiche 53, 1976.

[Pylyshyn75] Pylyshyn, Z.W., "Do we need images and analogs?" pp. 175-177, TINLAP-1, 1975.

[Rovner76] Rovner, P.D., "Automatic representation selection for associative data structures," Technical Report 10, Univ. of Rochester, Dept. of Computer Science, 1976.

[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.

[Sloman85] Sloman, A., "Why We Need Many Knowledge Representation Formalisms," University of Sussex, Cognitive Science Research Report, 1985.

[Smith85] Smith, D.E. and Genesereth, M.R., "Ordering Conjunctive Queries," Artificial Intelligence, 26, No. 2, pp.171-215, 1985.

[Stickel86] Stickel, M.E., "Schubert's Steamroller Problem: Formulations and Solutions," Automated Reasoning, 2, No. 1, pp.89-101, 1986.

[Subramanian87] Subramanian, D. and Genesereth, M.R., "The Relevance of Irrelevance," IJCAI87, pp.416-422, 1987.

[Weber83] Weber, K., How to Prepare for the New LSAT, Harcourt Brace Jovanovich, 1983.

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

CS-TR Scanning Project
Document Control Form

Date : 7/27/95

Report # AI-TR-1128

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 222(230-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other: Copy

Check each if included with document:
☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:

(A) IMAGE MAPS: (1-12) UNM'ED TITLE PAGE, UNM'ED BLANK, 1-3, UNM'ED BLANK, 4-8, UNM'ED BLANK, (13-222) PAGES #'ED 9-218, (223-230) SCAN CONTROL, COVER, SPINE, DOD(2), TRGT'S (3)

(B) COPY MARKS ON MOST PAGES.

Scanning Agent Signoff:

Date Received: 7/27/95 Date Scanned: 7/28/95 Date Returned: 8/3/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1128	2. GOVT ACCESSION NO. A210885	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Toward a Theory of Representation Design		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jeffrey Van Baalen		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1989 13. NUMBER OF PAGES 219
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

knowledge representation
knowledge based systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research is concerned with designing representations for analytical reasoning problems (of the sort found on the GRE and LSAT). These problems are intended to test the ability to draw logical conclusions from information presented and to synthesize that information in order to deduce interrelationships.

A computer program was developed that takes as input a straightforward predicate calculus translation of a problem, requests additional information if necessary, decides

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

what to represent and how, designs representations that capture the constraints of the problem, and finally, creates and executes a LISP program that uses those representations to produce a solution. Even though these problems are typically difficult for theorem provers to solve, the LISP program that uses the designed representations is very efficient.

The representations designed by the system are powerful because they capture the constraints of a problem in two ways. (i) The structure of the representation resembles the structure of the thing represented. For example, consider representing married couples as sets of size two. The structure of this representation resembles the structure of married couples because both have exactly two individuals in them. (ii) The structure enables efficient behaviors that enforce a problem's constraints by keeping them invariant in the structure. For example, as a set representing a married couple is manipulated, a behavior associated with it maintains its fixed size. This behavior efficiently enforces the size constraint on married couples.

The system designs a representation that captures as many of the constraints of a problem as possible. When a representation captures more constraints fewer sets of facts can be expressed in it. For example, in a representation that does not capture the symmetry of the married relation, one can state "A is married to B" and "B is not married to A." However, it is not possible to express these two statements in a representation that captures symmetry. Allowing fewer sets of facts to be stated in a representation reduces the space that a problem solver must consider; this in turn results in more efficient problem solving behavior.

Representation design consists of three processes: *classification*, *concept introduction*, and *operationalization*. Classification uses a library of structures each capturing different kinds of constraints. It finds the most specialized library structure for representing each concept in a problem. Concept introduction is a way of enhancing classification. When classification fails to capture all the constraints on a concept, introduction tries different ways to represent the concept. For example, sometimes when classification fails to capture all the constraints on a relation like "married," it introduces a concept like "couple," a function mapping an individual to the set of individuals he/she is married to. Classification of "couple" results in a representation that captures more constraints than did the representation of "married."

Classification and concept introduction run as coroutines, trying to capture all of the constraints of a problem. As they do this, the statements of captured constraints get removed from the problem. However, in many problems these processes fail to capture all of the constraints of a problem, leaving statements of the uncaptured constraints. Operationalization then tries to capture the constraints of the remaining statements by writing procedures and using these to further specialize the representations created by the previous processes.

The demonstration system has designed representations for twelve analytical reasoning problems. In each case, designing a representation and using it to solve the problem has proven to be far more efficient than using a binary resolution theorem prover to search for a solution in the initial representation.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 7/28/1995

M.I.T. Libraries
Document Services