NON-MONOTONIC KNOWLEDGE REPRESENTATION AND REASONING FOR NATURAL DESCRIPTION OF GOTHIC CATHEDRALS

by

TYLER GREGG CARLSON

(Under the Direction of Walter D. Potter)

ABSTRACT

The goal of the Architecture Represented Computationally (ARC) project is to transform user input or scholarly written descriptions of Gothic cathedrals into logical representations, allowing consistency validation, query-answering, and generation of precise descriptions or visualizations. The work of this thesis is the design and implementation of the first major step of this project, the ARC Logic system, which logically represents domain knowledge and performs inference on this knowledge. The ARC Logic system is domain independent; all domain information is entered by users through input methods, allowing the user to add terminology definitions, facts, and complex rules, without knowing Prolog. This system has a non-monotonic knowledge representation and inference engine, so the system can work with uncertain information, and fill in the information not explicitly stated with background knowledge, narrowing the gap between the logical knowledge representation and natural description in Gothic cathedrals and other real-world domains.

INDEX WORDS: Prolog, Gothic cathedral, Defeasible reasoning, Architecture, Non-monotonicity, Natural description, Domain-independence

NON-MONOTONIC KNOWLEDGE REPRESENTATION AND REASONING FOR NATURAL DESCRIPTION OF GOTHIC CATHEDRALS

by

TYLER GREGG CARLSON

BA, Wartburg College, 2009

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2012

© 2012

Tyler Gregg Carlson

All Rights Reserved

NON-MONOTONIC KNOWLEDGE REPRESENTATION AND REASONING FOR NATURAL DESCRIPTION OF GOTHIC CATHEDRALS

by

TYLER GREGG CARLSON

Major Professor: Walter D. Potter

Committee: Stefaan Van Liefferinge
Michael A. Covington

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2012

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1
INTRODUCTION

The goal of the ARC project is to allow users to easily create and use logical descriptions of Gothic cathedrals, and eventually to have an automated process by which usable logical descriptions can be extracted from natural language textual descriptions [2] [3]. This thesis comprises the development of the logical system that underlies the approach, called the ARC Logic system. The ARC Logic system, implemented in Prolog, contains a way to logically represent all the useful knowledge from a description or descriptions, and an engine to perform logical inference on this knowledge.

The first major component of the ARC Logic system is the ability to logically represent both the knowledge contained in a description and the knowledge of the domain as a whole. The domain of the ARC project is the description of Gothic cathedrals, and the knowledge representation of the ARC Logic system allows for description of Gothic cathedrals, in a way that is simple for the user, while also providing the functionality to make accurate and complete logical descriptions. To make a system that is both simple to use and fully functional, the ARC Logic system took cues for its design from natural descriptions of Gothic cathedrals and natural descriptions in general.

The ARC Logic system contains no information about Gothic cathedrals itself; it is domain-independent. Information about Gothic cathedrals in general and information about specific Gothic cathedrals are treated the same by the ARC Logic system. All of this information is called domain information, and it is all entered by the users of the system. This domain

information comes in only three forms, which are developed to coincide with the ways in which natural description would convey information about a cathedral or the domain of cathedrals. This domain information includes the defining of terminology for relationships, like "above," and objects, like "column," and ascribing logical properties to the words to match their implicit meaning. It also includes facts about objects and the relationships between those objects, like "this column has a base." Thirdly, the ARC Logic system has constraints, which are general and specific rules about objects in a cathedral and how they relate to each other, like "all columns have a base." With this domain information, the user can create descriptions of specific cathedrals, down to the smallest level of detail desired.

These three types of domain information are easy to use partially because they closely resemble the way information in conveyed in a natural description, but they are also easy to use because of their corresponding input methods. The ARC Logic system contains input methods the user calls to input their knowledge, about a specific cathedral or cathedrals in general, in the form of term definitions, facts, and constraints. These input methods allow the user to enter this domain information by following a simple syntax, and do not require the user to know Prolog or complicated programming principles.

The entering of domain information in the ARC Logic system is simpler, more efficient, and provides more functionality because the knowledge representation allows for the representation of non-monotonic knowledge. Non-monotonicity means that additional information can contradict and therefore remove already-known information, which opens the possibility for the users to work with assumptions when pieces of domain information, both facts and constraints, are uncertain. There are many approaches to non-monotonic reasoning, but the ARC Logic system is built on the concepts of defeasible reasoning. The system keeps track of

the certainty with which information is known, and can handle contradictions. This approach allows the user to enter domain information in a manner closer to natural description. The user can write constraints that apply in general, even if they are not true in all circumstances, and then write constraints that are exceptions to the other constraints. This functionality is used to fill in the gaps of a generally-sparse specific description with information about the domain in general.

The second main aspect of the ARC Logic system is the inference engine itself. The inference engine finds and explicitly asserts all the facts that can be inferred from the information provided, and enforces logical consistency. A key component to this is the custom call predicate, which is not only truth-preserving, but defeasibility-preserving because it keeps track of the certainty of information used to prove some conclusion.

Chapter 2 of this thesis provides the background on the ARC project, including the direction and goals of the research, the choice of domain, and components of the ARC project outside the scope of the work of this thesis. This chapter also explains how the ARC Logic system fits into the larger implementation goal. Chapter 3 illustrates the general flow of the ARC Logic system and provides a tutorial example of a user describing a single column. The knowledge representation and the user input methods are discussed in detail in Chapter 4. Chapters 2-4 eschew discussion about the non-monotonic nature of the knowledge representation and inference for the sake of simplicity. This non-monotonicity is introduced in Chapter 5, which discusses the relevant aspects of natural description, how this pertains to non-monotonic logic, and specifically to defeasible reasoning. Chapter 6 also amends the previously-illustrated knowledge representation to briefly explain how defeasibility of knowledge is represented. Chapter 6 also explains the custom call predicate, `d_call`, used by the inference engine, which extends the basic functionality of Prolog to prove goals while also determining the defeasibility

of the proof. Defeasible knowledge representation and inference is useful because the system can appropriately handle most situations with conflicting facts and constraints. Chapter 6 ends with an explanation of the way conflicting facts and constraints are handled by the ARC Logic system. Finally, Chapter 7 moves beyond the basic column example to explain how the ARC Logic system works at the full cathedral level and how default descriptions can be combined and used to create complete and efficient logical descriptions of Gothic cathedrals.

CHAPTER 2
THE ARC PROJECT

The Architecture Represented Computationally (ARC) project is a group venture researching and implementing an artificial intelligence approach to natural description of Gothic cathedrals. The goal is to automatically convert natural descriptions of Gothic cathedrals into machine-useable data, and then perform logical reasoning on this data in order to achieve a deeper and richer level of understanding of the cathedral described.

Gothic cathedral architecture is the focus of the ARC project for two main reasons. First, the logical nature of Gothic cathedrals and their descriptions makes them a fitting domain for research in logical representation of natural description. Second, implementation will be useful for architectural historians and others to produce and analyze logical descriptions, and to convert the vast and varied historical and current writings on these cathedrals into machine-usable information. This chapter briefly introduces both of these attributes of Gothic cathedral architecture domain and where this thesis project fits into the overall goals of the ARC project.

2.1 LOGICAL STRUCTURE OF CATHEDRALS

Architecture in general is a domain governed by rules and principles, many of which have been or can be formalized into logic descriptions. Describing architecture is possible with standard logical knowledge representation of axioms, facts, rules, and standard methods of inference [4]. While many domains can be modelled to some degree with these logical principles,

Mitchell's work shows the relative directness with which the real-world domain of architecture can be analyzed logically [4].

These principles are especially present and obvious in large comunal buildings, such as Gothic cathedrals. Because these types of buildings often follow basic principles of symmetry and modularity, with heavy repetition of elements, they are especially suited for description via axioms, rules and facts, and logical inference [4].

Because architecture in general and Gothic cathedrals in particular are relatively formal and logical domains, it is not surprising that descriptions often, possibly unintentionally, are written in a logical manner. One way this is done is by replacing repetitive explicit descriptions of similar elements with rules describing a pattern. Instead of saying "the first column has a base, the second column has a base, the third column has a base, etc. etc.," the description would often talk about columns as a concept, and that part of the concept of a column is that it has a base. To know that some particular column had a base would then require the use of inference. Finding all the explicit facts contained indirectly in the rule-heavy natural descriptions of Gothic cathedrals requires the use of a critic, or method for critical analysis.

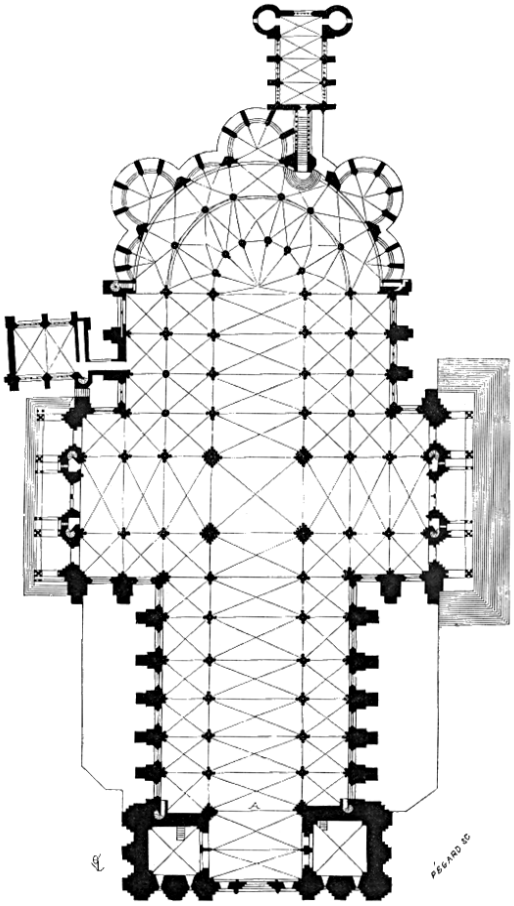

Figure 1. Floor Plan of Chartres Cathedral **[1]**

Mitchell describes the critic as consisting of three parts [4]. First, the critic has facts and rules relevant to the domain. Second, the critic has a set of observations about a design proposal. These two elements are stored together in a knowledge base like that in Prolog and other logic programming languages [4]. Third, reasoning with this knowledge base entails a set of true assertions about the design [4].

The implementation of the ARC Logic system generally follows Mitchell's description of a critical analysis system. The critic is the ARC Logic system itself, which provides analysis for descriptions of Gothic cathedrals. The design proposals are the natural language text descriptions about specific Gothic cathedrals. The set of observations of these design proposals is the information that is extracted from the natural language text and entered through the input methods of the ARC Logic system. The information in these input methods is transformed into Prolog facts and rules and entered into the database. Additionally, this database contains other facts and rules which make up the background knowledge for the domain. Though this is not necessarilly the case in Mitchell's description, in the ARC Logic system, the background knowledge is entered through the same input methods as observations from specific cathedral descriptions. When the inference engine is run on this knowledge base, all the facts that can be derived are explicitly dervied, so knowledge base plus inference engine entail all the true facts about the design [4].

The practical goal of the ARC project is to build a complete system by which user input or scholarly written descriptions of Gothic cathedrals are automatically transformed and represented logically. The ARC Logic system does not handle natural language text, but it takes input of term definitions, constraints, and facts, and converts these into correct Prolog facts and

rules. Once the information is in this usable form, inference can be performed, queries can be answered, and consistency of the description can be checked by the ARC Logic system.

The ability to answer queries is a useful tool for architectural historians, students, and anyone else interested in the architecture of Gothic cathedrals. Having descriptions encoded into Prolog allows users to interact with the description in a more dynamic manner, moving away from the experience of reading a book, and closer to the experience of interacting with an expert. The ARC Logic system allows the description to be queried to answer specific questions about the cathedral, can retrieve particular sections or aspects of the cathedral for examination, and can even compare entirely separate cathedrals against each other. Different descriptions of the same cathedral can also be compared to find patterns in style as they vary by author, time, or other factors, as well as be combined together to create more complete descriptions.

The ability to check consistency of the logical description is another valuable aspect of the system. The ARC Logic system can highlight or automatically correct inconsistencies within a description. The ability to identify problems in a description as it is being created is an important feature. This kind of feedback would be especially useful for students learning the structure of cathedrals, or professionals working on complex descriptions. Consistency checking could also be used to fix inaccurate or incomplete historical descriptions, highlight changes to a cathedral over time, or even help architectural historians translate descriptions from another language [2]. Consistency checking can also provide useful feedback for an automated natural language processing system. If the NLP system interprets some piece of text incorrectly, and it is entered as input to the ARC Logic system, if this incorrect interpretation leads to an inconsistency, the ARC Logic system can point this out or handle the conflicting information automatically.

2.2 DEFAULTS IN DESCRIPTION OF GOTHIC CATHEDRALS

Even though natural descriptions of Gothic cathedrals generally follow a logical structure, these descriptions can be very complex. This complexity arises because these descriptions often exploit human abilities of understanding that are difficult to computationally recognize and understand. A natural description of a Gothic cathedral can convey a large and complex logical structure with a relatively small amount of text. There are a number of reasons why natural descriptions are so efficient. First, this efficiency is partially accomplished by the human ability to combine propositions and leave out redundant identifying information. The sentence "the column has a capital and base" is a shortening of "the column has a capital and the column has a base." Since natural descriptions are written by human beings for human beings, the describer can make the assumption that the receiver will have the same understanding of the longer version. Sorting out the structure to clarify subject-predicate-object relationships is a general task for natural language processing. This same feature can apply over whole sections of a description as well. In natural descriptions of Gothic cathedrals, the subtle shifting of context is also common, such as talking about a cathedral in particular sections of the building instead of as a whole, or moving back and forth between descriptions of two cathedrals for comparison. This shifting context is also generally easy for humans to follow but difficult for natural language processing. The ARC Logic system includes a specific functionality called scope, which allows the user to set the context so that it does not have to be explicitly included in each piece of input. The use of assumptions within the text to write and speak succinctly by avoiding repetitive information certainly factors into the complexity of natural language processing, but it is not the major reason why a short natural description can represent a massive amount of logical information.

The principle reason for the efficiency of natural description is the use of assumptions about things that are left out completely. In a natural description of any scene, an overwhelming majority of the information needed to understand that scene is not explicitly present in the description. To understand any description requires background information, and the more common knowledge or experience shared between the describer and the person receiving the description, the more is left to implicit assumptions. Not only would two architectural historians talk about a cathedral with jargon that is unfamiliar to the layperson, they can also avoid explaining the basic rules of Gothic architecture to each other.

The ARC project requires capturing the information present in natural descriptions as accurately as possible in order to build the logical model for understanding the description. It was clear to the ARC research group, before the work of this thesis was started, that some way to fill in the gaps of the description, to use background information, in order to create a complete logical model was necessary. The domain of Gothic cathedral architecture is a good domain for exploration of these techniques. Though individual cathedrals can vary in quite significant ways, the concept of Gothic cathedrals has a default model, and the necessary background can be limited to a finite amount of useful information. Even to the layperson with minimal exposure to Gothic cathedrals, simply knowing that a building is a Gothic cathedral evokes many descriptive details of that building. The symbolic nature of the design, strict rules surrounding religious worship and the use of the space to facilitate this worship, and careful considerations for the laws of physics, are all likely factors in their adherence to a common model.

The specific implementation of the ARC Logic system allows the user to create default models, or use default models written by others, that fill in the gaps in descriptions. The ARC Logic system uses non-monotonic knowledge representation and reasoning, which lets the users

work with assumptions. The ability to make assumptions, which hold unless there is a reason not to hold them, creates a simple method for combining descriptions with background information.

2.3 ARC LOGIC IMPLEMENTATION

The ARC Logic system was designed to work as a usable standalone program, but also designed to be easily incorporated into a larger software package. Currently only the ARC Logic system is implemented, but work is underway with the ARC project for considerable extension of the ARC Logic system, on both the input and output side.

The ability to automatically extract the useful information from natural textual descriptions is a major goal of the ARC project. Deep and accurate natural language processing is extremely difficult, but the ability to do this well opens up many possibilities for examining Gothic cathedrals, and could also progress NLP research in general. Another goal of the ARC project is the ability to use precise logical descriptions of cathedrals to generate two or three-dimensional visualizations [2] [3], either as strict output or as an interactive system that allows a user to modify the logical model graphically.

While the eventual goal of the system is to allow for the automatic processing of natural language and output to visualization software, the system is designed to interface directly with users. Users enter information in a simple logical syntax, which is then converted into Prolog code, so that users can run inference on the system and query the knowledge base for results.

The ARC Logic system is designed with usability in mind, both for users working with the program as a standalone tool, and compatibility with other pieces of software, like natural language processing and visualization software. A main aspect of this usability is that the system is designed to be a black box; the specific implementation does not need to be understood to operate the system. Instead, the user or other software calls specific input methods to add rules

and facts to the Prolog knowledge base, and the output is a simple form that can be queried or transferred to other programs. The ARC Logic system is domain-independent; it has no built-in rules, facts, or terminology about cathedrals or any other domain. All information about the domain is entered through the limited set of methods, so there is no need for the user to alter the ARC Logic code unless they desire additional or modified functionality.

Not only can users remain ignorant of the inner workings of the ARC Logic code, they can be generally ignorant of the Prolog programming language and programming languages in general. Writing input methods requires following some basic syntax, but it does not require any understanding of Prolog or software engineering concepts. The input methods use basic logical properties and logical statements, which are automatically turned into correct Prolog code. This is a valuable feature both because of the implementation and the target user base. Prolog is especially suited for this kind of logical inference, but the language is not as widely-known as many "general purpose" procedural languages such as C or Java, and it works in a significantly different way than those procedural languages. Writing automated programs or interfaces to work with the ARC Logic system is made much simpler by the non-Prolog specific input and output methods. Even more basically, the ARC Logic software is designed to be used as a standalone product by architectural historians, students, and laypersons interested in architecture, some of whom have little to no experience with programming. The modular, black box, and domain independent nature of the ARC Logic system allow it to be integrated with other software as easily as it can be used directly by an end-user.

CHAPTER 3

EXAMPLE OF ARC LOGIC SYSTEM FLOW

This chapter will explain, in the form of a tutorial example, a very basic use of the ARC Logic system. Generally, the ARC Logic system takes input from the users, creates the proper Prolog rules and facts from this input, performs inference with those rules and facts, and allows the user to query the results. This chapter briefly discusses how this system works as a whole, and subsequent chapters explain the details of each individual component of the system. To demonstrate the functionality of the system in a general overview, this chapter will use a rather simple example of a single column (*Figure 2*). The ARC Logic system treats a single column in the same manner as an entire cathedral, so a single column can be used to illustrate the features of the program in a scaled-down example. Similarly, a software program, or multiple users, can interact with the ARC Logic system in the same manner as a single user, so for the sake of simplicity the example will assume there is a single user operating the ARC Logic system as a standalone application.

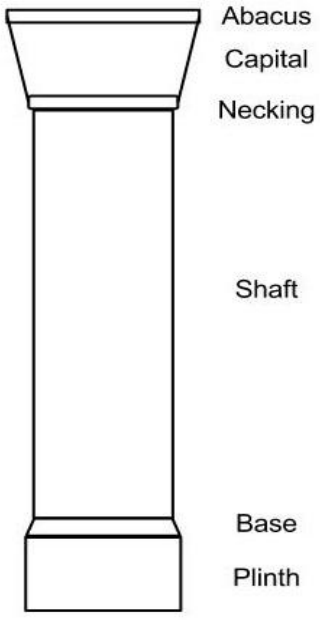

Figure 2. Example Column.

3.1 INPUT DOMAIN KNOWLEDGE

The first step for the user is to create the domain information to add to the ARC Logic system. Of course, information must be added to the system before it can be used for inference or querying, but there is no strict ordering of the "add information" and "use information" steps. Information can be added, inference can be performed on this information to create new facts, and then additional information can be added, some of which could contradict the earlier facts. This information comes in three main forms: term definitions, constraints, and facts. Each of these forms has corresponding input methods so that the user can input this information in a format that is understandable for those without programming knowledge. The information entered through the input methods is automatically transformed by the ARC Logic system into the appropriate Prolog rules and facts.

First, term definitions give the system a vocabulary with which to operate. Terminology specific to the domain is not built in to the ARC Logic system; the names and definitions for the relationships between objects must be defined with the user input methods. In order to create a useful complete description of a column, the user would describe the relations between the parts of the column. An obvious relationship would be "above." Relationship definitions let the user indicate that the ARC Logic system should treat above as a relationship. Definitions are more than just listing out words; they are a way to ascribe the desired behaviors to the relationships. For example, "aboveness" is transitive; if *a* is above *b* and *b* is above *c*, then *a* is above *c*. It is possible to express this semantic quality so that the ARC Logic system can derive information from the implied semantic value of terminology. This terminology is added, without any programming ability required, with the relationship definition methods. All it takes to signify

that "above" is a relation with the expected behavior is: define_relationship(above, [transitive]) (followed by a period).

The second type of information is facts. Facts include object facts like object(capital, 2) and object(shaft, 3), which mean that the unique ID of 2 represents an object of type capital, and 3 is an object of type shaft. There are also facts about relationships, such as above(2,3), which means that object 2 is above object 3. The user will likely add a majority of the information through definitions and constraints instead of asserting facts directly. The inference engine uses the term definitions and constraints to create most of the facts, and it does so by using the same fact assertion method internally. In the case of this column example, only the column itself needs to be created by the user. Object instances can be given a non-number name or else they are auto-assigned a unique ID number for a name. User-created names must be Prolog atoms, which begin with lower case letters or are in single quotes. In this example, the column will be name ex_column, and objects created by inference will automatically be given a number. The user calls the assert_object_instance predicate with the optional parameter of a name filled in, which looks like: assert_object_instance(column, 0, ex_column). The next chapter contains a detailed explanation of this method, including the use the second argument, which is the pre-existing object to which the new object belongs.

The user could describe ex_column by asserting additional details as facts. Since our example column includes a capital, a shaft, and a base, these three objects should be included in this example description. The user could enter this information in the same way as the column, by typing assert_object_instance(column, ex_column, ex_capital). The user could also directly assert relations by using the input method to say assert_fact(above(ex_capital, ex_shaft)) and assert_fact(above(ex_shaft, ex_base)). The user, however, does not need to

explicitly assert all these facts. Gothic cathedral architecture is especially suited for logical rules and inference because there are many repeating elements. The pieces that constitute the column and the relations between these pieces apply to every column, so making specific assertions for more than one column is redundant, and describing each individual column in an entire cathedral would be very tedious.

The ARC Logic system has a way to express the knowledge that certain types of objects always contain certain parts or that certain relationships hold between types of objects. This third and final type of information is called a constraint. Constraints can range in complexity, but expressing that every column must have a capital is done by create_constraint(column, must, has, capital). Additional constraints for shaft and base are created the same way. Instead of asserting facts for each user-defined relationship between object instances, the user can also write constraints for these. The user can say create_constraint(capital, must, above, shaft) and create_constraint(shaft, must above, base) to express those relationships. All the term definitions, facts, and constraints used in this example are shown in Appendix B, along with more complex examples.

3.2 INFERENCE

Inference is the second main function of the ARC Logic system. Prolog facts and rules are created automatically from the term definitions, constraints, and fact assertions, and are added to the knowledge base (KB), along with the operational code for the ARC Logic system itself. When the user tells the system to run the inference engine, every fact that can be inferred from the rules and facts is added to the KB. This inference engine continues until everything that can be inferred is in the knowledge base as an explicit fact.

Returning to the column example, assume that only the column was directly asserted, and the five constraints mentioned were created. Before the inference engine is run, the knowledge base (KB) has only the fact object(column, ex_column), and a number of rules created from the constraints and term definitions. The user will type inference and the inference engine will find everything it can from the facts and rules, and add them to the KB. Since there is a column fact in the KB, the constraints about columns are applied, and the inference engine automatically asserts a capital object, a shaft object, and a base object, and a has relationship fact for each showing they are part of ex_column. The column was given a name ex_column, but the assert_object_instance method will automatically assign a new object a unique number if a name is not given, which is what happens when the inference engine automatically asserts new objects. The creation of a capital, shaft, and base object, is all that is done in the first round of inference, but since the inference engine found some new facts, the inference continues another round automatically.

On this second round, since the KB contains capital, shaft, and base objects, constraints that apply to capitals, shafts, and bases will also be enforced. Because the user entered create_constraint(capital, must, above, shaft), the inference engine creates the facts for the above relationship between the capital and the shaft, like above(1,2). The same applies to the relationship between shafts and bases.

In the next round of inference, since there are new facts about the above relation in the KB, rules from term definitions about above are enforced. Because the user defined above as being transitive, and the KB has the facts above(1,2) and above(2,3), the inference engine also asserts above(1,3), signifying that the capital object is above the base object, even though a constraint was never written for this relationship, and it was never asserted as a fact.

The inference engine has to complete one full cycle without any new facts found so that it knows it has exhausted all possible avenues of inference. When the inference engine is done, all the information that can be derived from the user input has been derived, and the information is available to be printed in its entirety, queried, or used by some other software component. The information in the knowledge base can also at this point be added to or amended. A new call to the inference engine ensures that all new information is properly inferred and anything that is no longer known derivable is removed from the knowledge base.

3.3 QUERYING THE KNOWLEDGE BASE

After entering domain knowledge and running the inference engine, the knowledge base will be full of all the facts that can be inferred from the information the user entered. The final step is retrieval of this information. There are a number of predicates that can be used to retrieve all facts, all constraints, and all terms (see Appendix A), but generally a user would want to run queries on the data. A query is a set of conditions that have to be met, so any query-answering function will return either a piece of data that matches the query or indicate that there is nothing that matches the query.

The user might want to write a query to return all objects in the knowledge base, now that constraints have been added and inference performed. The user can simply write `q(object(X,Y))`. The capital letters are variables that values can match to. When the user presses the ENTER key, the first match is returned, and the display will look similar to:

```
?- q(object(X,Y)).
(indefeasible)
X = base,
Y = 1
```

For now, the (indefeasible) part can be ignored. The query found the first matching values in the KB, and shows the data that matches the X and Y variables. The output did not end with a period, signifying there are other possible results. Typing a semicolon signals the Prolog query to check for other possible results. This can be pushed multiple times until all results have been found, resulting in:

```
(indefeasible)
X = capital,
Y = 2 ;

(indefeasible)
X = column,
Y = ex_column ;

(indefeasible)
X = shaft,
Y = 3.
```

Queries are based on simple principles, but with these principles a user can easily write more complex queries. If the user wanted to query all the capitals, they could use the same query but with more information provided: q(object(capital,X)) would return X = 2. A query can be performed to find all objects that belong to ex_column by writing q(has(ex_column, X)).

The q method is over-loaded, so it works differently depending on the type and number of arguments. A query can be written with a second argument that is left as a variable (that is not already in use), and a set of all the matches will be returned as that variable:

```
?- q(object(X,Y), Z).
Z = [ (object(base, 1), indefeasible), (object(capital, 2), indefeasible), (object(column, ex_column),
indefeasible), (object(shaft, 3), indefeasible)].
```

Multiple clauses can be written in a single query. Any number of clauses, separated by commas, can be entered as a query as long as the query itself is surrounded by parentheses. This enclosing in a set of parenthesis is necessary for Prolog to treat the multiple clauses as one argument. The comma in Prolog is equivalent to the AND logical connective. Obviously, conjunctive clauses can be used to query with more precision. The user could for example write q((object(capital,X), has(ex_column,X))) to find all capital objects that are part of ex_column, but no other capital objects (if there were any in the knowledge base).

Additional clauses might also be added not to limit the query results, but to ensure the desired information is returned. In the example of q(has(ex_column, X)), only the names/ID numbers are returned. If the user wanted to know the object types corresponding to each of these ID numbers, the query could be written as q((has(ex_column, X), object(Type, X))).

The user can also query with disjunctions. Disjunctions are done in standard Prolog with a semicolon instead of a comma, and the q function of the ARC Logic system also handles the semicolon this way. The user can write q((object(capital, X) ; object(base, X))) to return for X every capital and base object. This disjunctive predicate is a 2-arity predicate, so everything before and everything after the semicolon must be a single clause or enclosed in parentheses.

In addition to the conjunctive and disjunctive logical connectives, the user can also query with negation. For example, the user may want to know every object that is part of ex_column, except capitals. This can be done with the predicate unless/1. The user could write q((has(ex_column, X), unless(object(capital, X)))), which will return the name for the shaft and the base in the example. Anything that can be written as a query can be written inside an unless clause; the same rule about enclosing within a set of parenthesis applies.

The comma is equivalent to logical conjunction and the semicolon to disjunction, but the unless function is not equivalent to the logic negation function. Standard Prolog does not have explicit negation. There is no way to prove something is false because Prolog works with Horn clauses, which cannot express negative information [5]. When a Prolog query returns true, it means that the goal was proven. When Prolog returns false, it means that the goal could not be proven. Using unprovability as a form of negation is called negation by failure, designated by a built-in predicate \+. Unless one is working with a closed-world assumption, where everything that can be known is already in the knowledge base, negation-by-failure is logically different than proving the goal is false [6]. A query of \+ some_goal could be true, and then be false when information is added to prove some_goal. The q predicate handles \+ the same way as unless, but the term unless, which can be read as "unless it is provable that," was chosen for this implementation to clearly distinguish this concept of negation-by-failure from explicit negation.

Figure 3 illustrates the system as a whole. Input queries are those that call input methods, which add facts and rules to the domain knowledge. The inference engine uses facts and rules of domain knowledge, but only creates new facts. Other Queries, such as q(object(X,Y)), return matching facts. Together this basic framework provides all the functionality of the ARC Logic system.

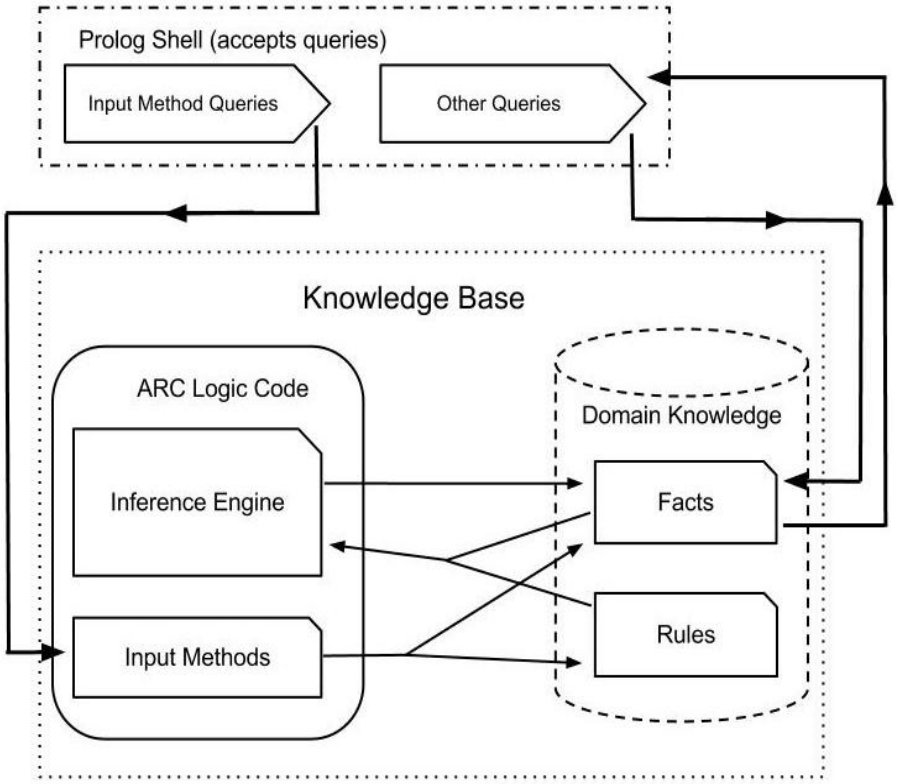

Figure 3. ARC Logic System Diagram.

CHAPTER 4

KNOWLEDGE REPRESENTATION AND INPUT METHODS

A useful knowledge representation is able to hold, in a structured way, all the information from the natural descriptions that is necessary for the logical description. Special concern was taken with the ARC Logic system to ensure that the knowledge representation is as close to natural description as possible while also being formal enough for inference. One way that it follows natural description is the three types of knowledge in this system, collectively referred to as domain information. Terminology automatically follows implicit logical behaviors, and constraints describe objects and the relations between them close to the way two human beings would communicate rules. The knowledge representation of the ARC Logic system also closes the gap between logical description and natural description by allowing for non-monotonic reasoning, which will be discussed in detail in Chapter 5.

The original conception for the knowledge representation of domain information for the ARC project was that domain information would be entered as Prolog rules such as [3]:

```
immediately_above(X,Y) :-
        necking(X),
        shaft(Y),
        has(ParticularColumn, X),
        has(ParticularColumn, Y).
```

While this approach allows those experienced with Prolog and intimate with the system the flexibility to create any rule that is acceptable in Prolog, it has a number of downsides. Most obviously, a user wanting to add rules would need to be proficient enough in Prolog to write a

rule in the correct syntax, ensure the rule actually reflects the intended semantics, and be aware of any problems, such as infinite loops, that could be caused by inference on that rule.

The ARC Logic system contains no domain information; all domain information is input data to the system. This feature has two main advantages: severability of the code from domain knowledge and control over the way domain information is encoded into the knowledge base.

First, the domain-independence means the file with the ARC Logic code is not messy with code for specific domain rules, and users do not need to alter the ARC Logic code in order to alter domain information. All domain information is entered as calls to a limited set of input methods. There are input methods corresponding to each type of domain knowledge, such as assert_fact and create_constraint.

Since all domain information is entered as data in the input methods, the ARC Logic system has control over the way in which domain information enters the knowledge base. The use of specific input methods means the ARC Logic system can enforce a standardized method for representing knowledge, instead of trying to use whatever specifications the user decided to use for each rule. The uniformity and predictability of the syntax of the Prolog rules is necessary for some of the more complex functions of the ARC Logic system, like constraint comparison, which will be covered in a later chapter. This standardization also allows for the knowledge representation to be designed in conjunction with the inference engine in order to avoid any possible loops from infinite recursion.

The use of set input methods simplifies the coding process extensively, but it also simplifies the use of the system for the end-user. The input methods are designed so that a user with no knowledge of Prolog, or programming in general, can assert facts, define terminology, and create constraints to match their own knowledge of the domain. A user need only follow a

simple syntax and have a basic understanding of logical properties, like transitivity, to build complex ontologies. This feature is made possible by the use of meta-programming techniques. Meta-programming, or programming programs, allows a program to process other pieces of the program like data, and turn data into part of the program, so that the new program can then process additional data. The nature of Prolog makes meta-programming easy, as there is no strict boundary between the program structure and the data. The knowledge base stores all the code for the program, but the knowledge base is also dynamic, and facts and rules can be added to or removed from the KB directly by the user or by other pieces of code. The input methods of the ARC Logic system take the data input to the arguments of an input method, and convert this information into Prolog rules and facts, which are then asserted to the knowledge base, so that they become part of the program itself. This chapter describes each of these three types of domain knowledge, how to use their corresponding input methods, and briefly how the data from these input methods are translated into Prolog facts and rules.

4.1 FACT ASSERTION

While the majority of fact assertions are made by the inference engine, the user can assert facts directly, and will at the minimum need to assert one object instance, such as a column or a cathedral. The term "fact" in general Prolog parlance is simply an entry in the knowledge base without an implication (:-). ARC facts are stored in the KB as Prolog facts, but they are not stored directly, like above(4,3) or object(column, 3). ARC facts are encapsulated by another predicate, fact/3 (meaning a predicate named fact with three arguments). The first argument of fact/3 is the fact itself, such as above(4,3) or object(column, 3), and the other two arguments

are metadata about the fact which are important for the non-monotonic nature of the knowledge representation.

The ARC Logic system has facts about relationships. The fact object(column,3) can be considered a combination of two separate facts: isType(3,column) and exists(3). In this implementation, all constants must be typed objects and if an object does not exist, there is no reason to ascribe it a type, so there is little need for severability of these two concepts. The syntax for object facts is meant to simplify the system for the user.

The input method for adding facts, assert_fact(+Fact,+Defeasibility,-Asserted), is used for both relationship facts and object facts, and is also used by the ARC Logic inference engine to assert every fact that is inferred. Clearly Fact here is the fact itself (like above(1,2) or object(column,1)). The concept of defeasibility will be introduced later, and the value from this argument becomes one of the two pieces of metadata about the fact.

The Asserted argument returns either a list with a fact in it, or an empty list. Just because the assert_fact method is called does not guarantee the fact will be asserted. One common reason for a fact not being asserted is that the fact is already in the knowledge base. The other possibility is a conflict is found when the fact is compared with the information that is already known. If the value in the Fact argument is actually asserted, that value is returned in Asserted. This functionality is important for the ARC Logic system to keep track of new facts, so that it knows when to continue and when to stop the inference engine. For user simplicity, assert_fact can be called with just one or two arguments: assert_fact(+Fact,+Defeasibility) and assert_fact(+Fact). Knowing whether or not the fact was asserted is generally not an issue for the user, and if Defeasibility is left out, it defaults to indefeasible.

Object facts are also asserted by assert_fact, but an additional fact must be asserted along with the object fact. For this reason, there is a special input method for objects: assert_object_instance(+Type,+IsPartOf,?Name,+Defeasibility,-Asserted). This method actually just calls two assert_fact methods: assert_fact(object(Type, Name)) and assert_fact(has(IsPartOf, Name)). A has relation must accompany every object because each object must be contained within something else. By default the root container is 0, but the user can designate the root container as 0, or the_world, or 'France', or whatever the user wants. The value given for IsPartOf, such as 0 or 'France', does not need to be an object, and of course the top-level container(s) can't be an object, because each object needs to be part of something else.

The Type argument is of course the object type of the object to be created. Defeasiblity and Asserted works the same way, except Defeasibility applies to both facts and Asserted is a combination of the Asserted values from the two assert_fact methods it calls.

The Name argument can be instantiated or left as a variable. If it is instantiated, then the object is created with whatever atom is given. While there is good reason to treat proper names as values tied to the real constants [7], it is unlikely a user would want to name an object after it has been created, so in this system names are used directly for instances. If the user does not name the specific object, or create_object_instance is called internally by the inference engine, the object is given a unique ID number, which is matched to the Name argument.

The identification of objects by unique number constants or names is a change from previous conceptions by the ARC project [2]. Originally, the idea was that object instances could be given a proper name, which is allowed in the ARC Logic system, or would be Skolemized [2]. Skolemization replaces an existential quantifier with a Skolem function that returns unique names for each instance [2]. The use of Skolemization in the ARC project creates a hierarchical

system that served as an address for a specific instance of an object. A capital for example could look like: base_inst(column_inst(arch_inst(nave_inst(cathedral_inst, 1), 1), 6), 2), 1), meaning the first base of the second column in the sixth arch in the first nave of the first cathedral [2]. This approach has the benefit of the constant being an address. Instead of a unique but arbitrary number, it is more clear which real world object this constant is mapped to. All of this information is available, however, in the ARC Logic system in a more flexible way; a basic Prolog function can be written to trace the "address" of a unique ID number back through its has relations, which can be formatted in whatever manner is desired. Also, the query q(contains(X, ex_base), object(Y,X)), would return all the objects (with their types) up to the top level which contain the object named ex_base.

The conception of Skolemization for the ARC project was used to enforce existence of objects belonging to another object. This duty is taken care of with constraints, which allow the user to state that an object of some type must have any number of objects of another type inside it. Individual Skolemization functions are replaced with the single function that returns unique ID numbers by incrementing each time a new number is needed. This process also keeps the system more in line with traditional database structures that have unique keys, which also adds to the usability and versatility of the system.

4.2 TERM DEFINITION

In order for a user to develop an ontology for some domain, they must first establish a vocabulary. In the ARC Logic system, this vocabulary is necessary for the relationships that hold between constants and for object types. Relationship terms must be defined explicitly before they can be used correctly. Object types can be used without being defined, but explicit definitions

add extra versatility. There are three definition methods for inputting information: define_relationship/2, define_metarelationship/3, and define_object/3. Each of these predicates takes the name or names of the relationships/objects being defined, and the logical properties that apply, and asserts all the necessary Prolog rules to the knowledge base at run-time.

4.2.1 Relationships

The Prolog rules for relationships can be created with define_relationship(+Name,+List). This predicate requires a Name for the relationship and a List of logical properties. The syntax for a Prolog list is brackets surrounding a number of items, which are separated by commas, such as [item1, item2]. For each of the logical properties in the list, one or more Prolog rules are created. There are three distinct logical properties, and the list can contain any number of these properties. If a relationship has no logical properties, an empty list is used, which is just the two brackets [ ]. The three basic logical properties that can appear in the list are symmetric, reflexive, and transitive.

If a relation includes the attribute symmetric, the following rule is added to the knowledge base, where name is the value given for the Name argument:

```
name(B, A) :- d_ground(name(A,B)).
```

If a relation includes the attribute reflexive, the following rules are added:

```
name(A, A) :-
        d_ground(name (A, _)).

name (A, A) :-
         d_ground(name (_, A)).
```

The reflexive rule could be applied in general with the fact '*name*(A, A).', but it is being used in a more limited way. What these rules say is that if some object has any relation of type *name* to anything, it has that relation to itself as well.

If a relation includes the transitive attribute, the following rule is added:

```
name (A, C) :-
        d_ground(name (A, B)),
        name (B, C).
```

Each of the rules made for logical properties of relations include the predicate d_ground, which is not built in to standard Prolog. The d_ground predicate does not have its own rule or fact anywhere in the knowledge base; it is a signal for d_call, a custom call predicate that will be described in detail in a subsequent chapter. d_ground(X) signals d_call to ensure that X is a fact in the KB. This approach puts a depth limit on d_call to ensure the avoidance of infinite loops. Each of the relationship definitions is recursive, and recursive rules are susceptible to entering infinite loops. If d_ground was not used, Prolog would try to prove the some_symmetric_relationship(X, Y) by proving some_symmetric_relationship(Y, X). To prove some_symmetric_relationship(Y, X), Prolog would use the same rule and the new goal would again be some_symmetric_relationship(X, Y). There are a number of approaches to cycle detection, such as keeping track of past goals, or space-saving variants of this. The ARC Logic system handles infinite recursive loops, and other issues, by having an inference engine that loops, so that not all steps have to be taken at once. Because of this, d_ground can be used, and all the facts can be found even though it moves only one step at a time. For example, if the KB includes above(1,2), above(2,3), and above(3,4), then the inference engine will find, because of the transitive rule of above, above(1,3) and above(2,4) on the first round of inference and above(1,4) on the second round.

There are two types of variations on the logical properties. One of these variations is defeasible versions of the properties, which will be discussed in a subsequent chapter. The other variation is negated versions of the properties. As explained in the previous chapter, Prolog does not have explicit negation. Because explicit negation is not allowed, the system does not have the

full power of first order logic [5]. Negation-by-failure (the unless predicate) is very useful for the system, and the functionality requires almost no additional cost [5], but for the ARC Logic system to have the desired functionality, it needs explicit negation. Explicit negation is done with a rather simple trick of creating regular facts that are marked as negative. These facts, such as not_above, are indistinguishable from other facts, like above, to the Prolog engine, but they are treated as negative by the ARC Logic system and the user. A simple predicate negate/2 interchanges a predicate between the positive and negative forms by adding or removing a not_ from the front of the name.

A user can include any of the negative versions of the logical properties: asymmetric, irreflexive, and intransitive. Negative rules are created the exact same way as their positive counterparts, except the name used for the head is changed to the negative version. Instead of the negated relationship being their own completely separate predicate, the negated versions are dependent on the existence of positive relation facts.

Asymmetric:

```
not_name(B, A) :- d_ground(name(A,B)).
```

Irreflexive:

```
not_name(A, A) :-
    d_ground(name (A, _)).

not_name (A, A) :-
    d_ground(name (_, A)).
```

Intransitive:

```
not_name (A, C) :-
    d_ground(name (A, B)),
    name (B, C).
```

The column example from the previous chapter can now be extended. The user could add the relation immediately_above and ensure it is treated as explicitly intransitive by entering

define_relationship(immediately_above,[intransitive]). If the knowledge base contains the facts immediately_above(a,b) and immediately_above(b,c), then the inference engine will assert not_immediately_above(a,c) to the knowledge base.

A user could work with a taxonomy that avoids explicit negation altogether. If immediately_above, for example, is created without any version of transitivity, the program would still not be able to infer immediately_above(a,c) in the above scenario. Adding intransitive to immediately_above, just like adding asymmetric and irreflexive to the above relation definition, is not necessary, but will increase the thoroughness of the procedure for checking consistency.

4.2.2 Meta-relationships

In addition to defining relationships between constants, the user can also define the relationships between these relationships with define_metarelationship/3. The first argument is either antonym or implies, and the next two arguments are previously-defined relationship terms.

Implication allows the user to express if-then relationships between the relationships. define_metarelationship(implies, immediately_above, above) allows the system to infer above(a,b) from immediately_above(a,b), but not the other way around. With the definition of immediately_above from the previous section, and this meta-relationship, instead of writing the constraints create_constraint(capital, must, above, shaft) and create_constraint(shaft, must above, base), the user can be more exact by writing create_constraint(capital, must, immediately_above, shaft) and create_constraint(shaft, must immediately_above, base). The ARC Logic system will then infer that the capital in the example is immediately above the

shaft and the shaft is immediately above the base. Once this information is in the knowledge base, the inference engine will find that the capital is above the shaft and the shaft is above the base. It will then infer that the capital is above the base, but not immediately above the base.

Antonymy in this implementation designates a converse relation, such that define_metarelationship(antonym, above, below) allows the system to infer below(b,a) from above(a,b), and vice-versa. Unlike explicit negations, the antonym of a relation is not an atom manipulation function, so the antonym pairs are actually stored in the KB and are used when checking fact assertions and constraint creations for conflicts. Synonymy is another logical meta-relation, but since it only creates a simple redundancy, the user, or a natural language processing system, should decide on one naming convention for each relationship.

4.2.3 Built-In Relationships

It should be clear that even though the ARC Logic system is domain-independent, the system makes some assumptions about the domain. Object types and hierarchies are built into the system because they are especially useful for describing architecture and similar domains. For objects to work correctly in the ARC Logic system, two special relationships are built into the system.

The has relation is built into the inference engine, because it works differently than user-defined objects, and is necessary for the connection between objects. The has relation represents an immediate containment (parent) of one object by another. The next section explains how the has relationship is used for checking the conditions of constraints, and how the constraints with has as their relationship work differently than those which have a user-defined relationship like above.

The has concept is also extended by the contains relationship. Each object contains itself, as well as all objects that it has, and all objects those objects has, etc. etc. The contains relation could be created with define_relationship(contains, [transitive, reflexive]), and define_metarelationship(implies, has, contains), but it is implemented differently. Creating the contains relation like a normal relationship means that contains facts are added to the KB. This would add a lot of likely unnecessary explicit facts to the KB, and the creation of these facts, which makes one transitive inference at a time, means the system has to do many loops of the inference engine, which can extend the time of the inference significantly. Instead, the special relationship between has and contains is exploited, so that contains has a non-looping transitivity rule like the classic ancestor-parent problem [6].

4.2.4 Object Definitions

Objects are not created with semantic value from logical properties, like relationship terms are. The system could have been designed where objects have implicit semantic value; it seems very reasonable after all to include in to the definition of a column that it is made of a capital, a shaft, and a base. Term definitions however, are reserved for those behaviors that are implicit in the word itself; rules such as "every column has a capital" is possible with constraints. Because all the information that might be implicit in an object type can be entered by constraints, there is no need to define objects before using them, like there is with relationships.

The define_object method is actually a variation on the define_metarelationship method, as it defines the relationships between object types. The define_object method allows the user to designate supertypes and subtypes of objects, which work similar to the implies meta-relationship. For example, in Gothic cathedral architecture, there are two types of supports:

columns and piers. Columns and piers share many features, so the taxonomy of architecture uses a general type, support, to apply to both. Though the column example from the previous chapter is about a single column, the user may want to ensure that the features which are shared between columns and piers are written as features of supports, so that in the future, the user can add a pier and not have to rewrite the same constraints for piers. To tell the ARC Logic system that a column is a type of support, the user simply writes define_object(subtype, column, support) or define_object(supertype, support, column). This information is kept in the KB like antonyms. Whenever any Prolog rule applies to supports, it must apply to columns and piers, but rules applying to columns or piers are not applied to supports. Instead of create_constraint(column, must, has, shaft), the user could say create_constraint(support, must, has, shaft). Since the ARC Logic inference engine knows that column is a type of support, it will make sure that the ex_column has a shaft.

The subtype relationship is the same as the implication meta-relationship in theory, but is implemented in a very different way. If the user inputs define_metarelationship(implies, immediately_above, above), and immediately_above(1,2), then above(1,2) is also inferred. Having both of these relationship facts in the KB is not problematic at all. Object facts, however, are a combination of a fact about the type and a fact about its existence. It would be acceptable to say that object 3 is of type support and type column, but it is problematic to say both a support and a column of the same number exist. In this implementation each existing constant corresponds to one and only one real world object. Instead of creating a rule like implication, the subtypes are noted, and whenever d_call requires the proof of an object, it will look for the proof of that object, or any subtype of that object. When d_call has to prove the goal of a support, it can do so with a fact for a column. This means the user can write constraints with supertypes in

the conditions, and they will apply to any subtypes. The query function also uses d_call, so any queries for supports will return relevant information about columns and piers. A user does not need to use any type hierarchy and inheritance; this feature is just for efficiency, and brings the logical description closer to natural description.

4.3 CONSTRAINT CREATION

The final type of domain knowledge used by the ARC Logic system is constraints. Constraints are statements that enforce existence of objects or relations between objects. When create_constraint is called, its arguments are automatically transformed into Prolog rules which are asserted to the knowledge base.

Like term definitions, constraints are designed with the intention of striking a balance between versatility and usability for those without Prolog experience. Like term definitions, which require understanding basic logical properties such as transitivity and implication, constraint creation also requires a basic use of logical concepts. The syntax for create_constraint falls between writing a First Order Logic sentence and writing out the intended rule naturally. The complete create_constraint/7 method has seven arguments, but this method, like most of the input methods in the ARC Logic system, is overloaded, and the user will often be able to enter a create_constraint with lower arity. When an input method with lower arity is used, it calls the complete version, with default values filled in. Appendix A shows the different forms of create_constraint that are available, and what default values are used.

In the overview chapter, one constraint used was "every column has a capital," which was added to the system with create_constraint(column, must, has, capital). This 4-arity version of create_constraint calls the full version which looks like:

create_constraint(X, object(column, X), must, has, 1, 1,capital), which can be read as: "For all X: if X is of type column, then X has a minimum and maximum of one capital."

The constraint follows the basic structure of a First Order Logic sentence. The first argument is a universal quantifier and the main connective is a conditional (if-then), which is the common accompaniment for a universal quantifier. If the antecedent is matched, then the inference engine also enforces the consequent, by asserting facts about objects and relations.

The full constraint begins with a universal quantifier. The user can enter any variable for this argument, but must ensure that the same variable is used in the second argument to designate what the constraint applies to. The second argument is called the condition clause, and can be considered the antecedent of the conditional connective. When an object type is the only condition, this clause is very simple, but the condition clause can be expanded with all the same complexity as is possible with a query, including conjunction, and unless. This similarity between the condition clause and a query is not a coincidence. The condition clause serves the same function as a query, in that it selects all and only the information in the knowledge base that matches. The only difference between the condition clause and a query is that the user must ensure the variable for the universal qualification clause matches the variable in the condition that will be matched with the constant (object instance) that the consequent of the constraint applies to. For example, if the user wanted to write a condition that only applied to columns that are contained inside the arcade (the lowest tier of a Gothic cathedral), they could write the condition as (object(column, X), contains(Y, X), object(arcade, Y)). It is important that the first argument of the constraint, the universal quantifier, is X. If the value of the first argument was set to Y, the constraint would instead apply to all arcades that contain a column.

The antecedent part of the constraint works the same no matter what the consequent of the constraint is. While the antecedent looks and acts like part of a first-order logic sentence, the rest of the arguments of create_constraint/7 are meant to follow a closer-to-natural-language syntax and use. This is advantageous for usability, as writing "there exists exactly one" or "there exists a minimum of 3 and a maximum of 6" becomes more complex in FOL. Instead, the constraints have more limited functionality, but are easier to understand and write.

The fourth argument of create_constraint/7 is the relation clause. As explained in the previous section, the built-in has relationship works slightly differently than the user-defined relations, like above. The main difference is the head of the rule that is created by the two kinds of constraints.

When the user enters create_constraint(column, must, has, capital), it is translated into:

```
object_to_assert(capital, A) :-
    label(A, [object(column, A), contains(0, A)], indefeasible, has, 1, 1, capital),
    [object(column, A), contains(0, A)],
    indefeasible,
    unless( min_is_met(A, has, 1, capital, [])),
```

The second line (starting with label), the second item in the list in the third line, and the fourth line can be ignored for now.

It should be clear that the first part of the third line is taken directly from the condition clause. Recall that the inference engine queries the knowledge base to find every fact that it can, and then attempts to assert these facts. In order to prove the goal of object_to_assert(capital, A), the call predicate needs to prove every clause in the body. Since the condition clause from the constraint is a clause in the body, the call predicate needs to prove this first. If the call makes

it past the conditions, then the variable A has been matched to some appropriate constant. Working with the same example knowledge base that has been used so far, the variable A gets matched to the constant ex_column.

The other clause that must be proven is the unless(min_is_met(...)) clause. min_is_met counts the number of objects of the type matching the seventh argument (capital in this case) that are had by the constant (ex_column in this case) selected by the condition clause. If min_is_met finds at least the minimum number of these objects, then it succeeds. If min_is_met succeeds, then unless(min_is_met(...)) fails. This means that if ex_column already has a capital object, the goal of object_to_create(capital, ex_column) fails, so no new capital object is created for ex_column. If min_is_met fails, then that means there are not enough objects to meet the minimum, so unless(min_is_met(...)) succeeds.

If all the clauses in the body of the rule succeed, then the instantiated head of the rule is returned to the inference engine as a success, and the inference engine then tries to assert that head, and all other goals it could prove, with assert_fact. The head of this rule, object_to_assert(capital, A), is a special designation so that the program calls assert_object_instance(capital, A), which then calls assert_fact for the two appropriate facts.

Constraints can be written for any user-defined relation as well, and though they work almost the same way, there are a few notable changes.

create_constraint(capital, must, above, shaft) is translated into:

```
above(A, B) :-
    label(A, [object(capital, A), contains(0, A)], indefeasible, above, all, all, shaft),
    [object(capital, A), contains(0, A)],
    indefeasible,
    unless(min_is_met(A, above, all, shaft,  [])),
    find_object_target(A, above, shaft, B).
```

Clearly the first change is the head. The goal for user-defined relationship constraints is slightly more straightforward. If the goal is proven, then the inference engine attempts to assert it as a fact.

The second change is the way that the minimum clause works. In user-defined relationship constraints, the value for minimum and maximum can be, and will default to if not given explicitly, the value all. The all signals that the relation holds for any number of objects matching the conditions. When the minimum value is all, the min_is_met goal always fails, so the unless(min_is_met(…)) goal always succeeds. If the minimum value is a number, the min_is_met clause works in a similar way to has constraints. The difference is that instead of counting "children" objects like the has relation does, it looks for "sibling" objects of the correct type and that have a fact for the relationship between the two, which essentially looks like: (has(X, Sibling1), has(X, Sibling2), above(Sibling1,Sibling2)).

Finally, the non-has constraints have a final clause of find_object_target/4. This is used to find a sibling object of the correct type, but where the relationship between the two objects is not already asserted to the knowledge base. find_object_target fails when there are no more objects of the necessary types to create relationship facts for. This happens if constraints determine that a column only has one shaft, but another constraint says the capital is above two shafts. This constraint will also inevitably fail when the all amount is used for minimum and maximum, because this ensures that it applies to each object it could apply to no matter the quantity. If the user-defined relationship constraints created new objects, the use of all as a quantity would not make any sense, just as it makes no sense for has constraints.

With constraints, term definitions, and the ability to assert facts, the user has all the basic functionality of the ARC Logic system. The system could be used, as it has been described so far,

to do classical logical inference, where every proposition is either true or false. The next chapter introduces, and the rest of this thesis is built on, aspects of the non-monotonic knowledge representation and inference that moves the system past classical logic and closer to natural description.

CHAPTER 5
NON-MONOTONICITY

The goal of the ARC project is to allow users, or natural language processing software, to easily create and use logical descriptions of Gothic cathedrals. The closer the logical system is to encapsulating the natural style of description, the more efficient and accurate it can be. For this reason, the ARC Logic system was designed with considerations of natural description, and the non-monotonic nature of the program addresses a significant element of natural description. Chapter 2 briefly explained that much of the efficiency of natural descriptions is a result of all the things that are not said explicitly, but are implied. Natural description relies heavily on assumptions and the ability to logically model this requires some way of working with assumptions. This chapter begins with a section providing background on the way natural description is non-monotonic. The second section explains the concept of monotonic and non-monotonic logics directly. The final section briefly explains defeasible reasoning, and how this is used by the ARC Logic system.

5.1 NATURAL DESCRIPTION

Natural descriptions do not exist in a void; they are informed by all the other relevant information related directly or indirectly to what is being described. Description of a formal domain, like mathematics, can start with a finite set of axioms and move from there. Outside of contrived domains, description of a domain is significantly more complicated. Information about

the domain can come from many, possibly conflicting, sources and methods over a long period of time. Also, determining the boundaries of information in a given domain is messy, as domains all bleed together, and can be related in complex webs of hierarchical relationships.

A natural description of any object or scenario in a real-world domain will likely explicitly say very little in comparison to all the information that is assumed. A typical natural description is not only light on explicit description for the sake of efficient communication; verifiable information that fits implicit assumptions is likely to go completely unnoticed because of the inclination to ignore consistent attributes. Someone may describe a Gothic cathedral, or any building, without any explicit mention of the floor. If the describer was asked about the cathedral having a floor, the answer would likely fall along the lines of "Of course it did! If it didn't have a floor, I would have noticed and said something about it."

Natural descriptions take into account the assumptions from generic models or sets of norms, and tend to only describe how a specific case varies from these generalities [8]. To model the way natural descriptions takes these assumptions for granted, the ARC Logic system allows users to create this background information by making general constraints about columns, naves, or entire Gothic cathedrals, and then allowing more specific rules to override the general rules. This method requires the ability to hold rules that apply to a set of constants (object instances), while holding conflicting rules that apply to some subset of that set, and then reason with all of these rules appropriately. In classical logic, contradicting information cannot be handled, so overgeneralizations cannot be made. Instead, each possible exception needs to be made explicit so that no rules could lead to both P and $\sim P$. The ARC Logic system allows users to write overgeneralizations and exceptions, which requires a different type of knowledge representation and reasoning than is provided by classical logic.

5.2 NON-MONOTONIC LOGICS

Monotonicity, an attribute of classical logic, is insufficient for modeling natural descriptions, so the ARC Logic system incorporates a non-monotonic knowledge representation. Monotonic functions are those that only increase or only decrease as the input to the function is increased or decreased. As the input to a monotonic function increases, there are only two possible behaviors for output: either the output is non-decreasing throughout, or it is non-increasing throughout. In monotonic logics, the entailment function is monotonic, so new information can only mean additional sentences, or nothing new, can be entailed. Nothing can be taken away that was previously entailed. If some set of sentences *A* entails a sentence *p*, then *p* is entailed by the union of *A* with any additional sentences.

Describing Gothic cathedrals monotonically would be very unnatural. If one were to make a statement about Gothic cathedrals in general, such as "Gothic cathedrals have four levels/tiers of elevation in the nave," then this rule, plus the fact that "Chartres is a Gothic cathedral," would entail "Chartres has four levels of elevation in the nave." However, human beings create rules for generalizations which do not apply to every scenario. Natural description, like human reasoning in general, can use imperfect knowledge to draw reasonable and useful conclusions. A fact could come from direct observation, such as "Chartres does not have four levels of elevation in the nave." These are two contradictory facts, and classical logic has no way to meaningfully handle such contradictions.

Natural descriptions often involve making generalized statements that do not always hold for every case. This is acceptable because humans are not as incorrigible as monotonic logic. Instead, most knowledge is held in a state of "X can be considered true unless there is a reason not to believe X." Based on the general 4-level rule, someone would assume Chartres has four

levels, but would likely have little trouble reconciling this assumption with additional information about the specific cathedral. For the ARC Logic system to have this ability, it must be able to reason non-monotonically and handle contradictions, which inevitably arise when using uncertain information. There are many different non-monotonic reasoning systems, but each has the ability to work with uncertain information. To use uncertain information and handle contradictions accordingly, non-monotonic systems have some way of keeping track of certainty of facts and rules.

5.3 DEFEASIBLE REASONING

Defeasible reasoning is one of the non-monotonic methods of logical reasoning. In defeasible reasoning, there are defeasible and indefeasible reasons. Indefeasible reasons logically entail their conclusions [9]. Indefeasible rules are often called strict rules because no exceptions can be made to the rule, and under no conditions could information from strict rules be contradicted. If just indefeasible premises are used, the logic system would be classical logic. Defeasible reasons do not logically entail their conclusions. Instead, they provide a reason to believe some conclusion, but allow for the possibility that some additional information could give a reason not to believe that conclusion. Defeasible reasoning is useful for logically modeling assumptions. The set {*P*, "defeasibly $P \rightarrow Q$"} d ⊨ (defeasibly entails) *Q*. If both of the sentences in the set are true, one should assume *Q* unless some additional information provides a reason not to.

The concept of "assume *Q* is true unless there is reason to believe otherwise" should sound familiar. Recall that Prolog's form of negation, negation-by-failure, is simply a measure of whether or not something can be proven true. If some_goal cannot be proven true, then \+

some_goal, or unless(some_goal) is true. Since this implementation, like many implementations, operates in an open world scenario, new information could be added to the knowledge base that proves some_goal is true. Prolog therefore has a built-in implementation of defeasible falsity. The next chapter will explain that in practical usage, implicit defeasibly false is still different than explicit defeasibly false, because the explicit version cancels out something being defeasibly true. The use of negation-by-failure as an implicit defeasible falsity is very useful in this implementation, but the ARC Logic system also requires a way to use defeasible reasoning to show something is defeasibly true.

The ARC Logic system implements these principles of defeasible reasoning in a manner specific to the aim of modeling natural description. The ARC Logic system uses the concept of defeasible knowledge as its measure of certainty. A fact in this implementation is indefeasibly true, defeasibly true, defeasibly false, or indefeasibly false. Rules are either defeasible or indefeasible. Some types of non-monotonicity have a scale of certainty of information. In probabilistic logics for example, a proposition *q* could be true with 87% likelihood. Defeasibility in the ARC Logic system is not a measure of how certain some information is, or how much credence to give some fact or rule; it is simply a binary of either known with certainty or not. This binary certainty level allows for reasoning to work very similarly to classical logics, as will be explained in the section on the certainty-preserving call predicate.

Defeasibility has no measure for deciding under what conditions something should be assumed. If the user decides that *P* should be assumed, and $P \rightarrow Q$ should be assumed, then the ARC Logic system will also assume *Q*. The ARC Logic system makes no determinations about the adequacy of the justifications for an assumption. Determining this would be extremely complex; it would involve determining the credibility of sources of information, where the

burden of proof lies, and many other factors. The justification for assumption is therefore left to the user. If a user adds a general rule to the ARC Logic system that says "all Gothic cathedrals have one fire-breathing dragon," the ARC Logic system will create a fire-breathing dragon instance for each Gothic cathedral instance unless it has a reason not to. This is of course no different from classical logic. {*cathedrals have dragons*, *Chartres is a cathedral*} ⊨ (entails) *Chartres has a dragon* in classical logic. Logic reasoning is truth-preserving; true arguments must lead to only true conclusions. Defeasible reasoning in the ARC Logic system is truth-preserving and defeasibility-preserving; indefeasibly true premises lead to only indefeasibly true conclusions, and reasonably-assumed premises lead to reasonably-assumed conclusions. The ARC Logic system has a simple but precise method for determining the defeasible truth value of a conclusion that directly parallels classical logic. This defeasibility-preserving inference is explained in the next chapter. The ARC Logic implementation of defeasible knowledge is only a minor expansion from classical logic, but it is enough of an expansion to allow the desired natural description ability of using overgeneralizations and exceptions.

The concept of *P* as a defeasible reason for *Q* is rather straightforward; if *P*, then assume *Q*, unless there is reason not to. It is the "unless" clause, the manner in which it is determined if *Q* is defeated, that adds complexity. Defeaters are types of reasons that do not allow a new conclusion to be drawn, but instead give reasons to not believe some assumption. John Pollock famously identified two different kinds of defeaters for defeasible reasons [9]. A rebutting defeater gives a reason to deny the conclusion, *Q*. An undercutting defeater attacks the reasoning, 'if *P* then defeasibly *Q*', itself, and therefore the deriving of *Q* from it [9]. The ARC Logic system uses, with liberties, the concept of rebutting and undercutting defeaters to compare conflicting constraints, which will be explained in detail in the next chapter.

Unlike a general extension of Prolog for defeasibility, like d-Prolog [10] [11], the ARC Logic system can take advantage of the general domain and the specific type of description used. In a defeasible reasoning system, two separate lines of inference can lead to conflicting conclusions. The most common example of this is called the Tweety triangle [10]. In the Tweety triangle, the following information is in the knowledge base:

bird(X) -defeasibly→ flies(X)
penguin(X) -defeasibly→ not_flies(X)
penguin(X) → bird(X)
penguin(tweety)

This situation allows for both not_flies(tweety) and flies(tweety) to be derived. When a conflict arises, argumentation is necessary. Argumentation is a way to reason about a claim based on the arguments for and against it [12]. Deciding which rules take priority over other rules can be complex. The domain of natural description makes this prioritization much easier. In the example of three and four storied Gothic cathedrals, it was intuitive and obvious that the person trusted the information specifically about Chartres over the conflicting information about Gothic cathedrals in general. There is a natural and useful bias to assume that more specific information takes precedence over less specific information. It is a justifiable assumption of the ARC Logic system that specificity is useful for choosing which piece of conflicting defeasible information to use.

The d-Prolog system makes the same assumptions about specificity as a natural way to resolve conflicts between defeasible rules [10]. d-Prolog determines specificity by checking if one chain of reasoning entails another. With the indefeasible information that penguin is a type of bird, the system concludes the rule saying penguins do not fly is more specific than the rule

saying birds fly, so this line of reasoning takes precedence and it is determined that Tweety does not fly.

d-Prolog is written for very general logic programming. The ARC Logic system has the advantage of a specific domain, so alternative approaches for determining specificity can be used. The domain of description as implemented in the ARC Logic system has a very straight-forward way of measuring specificity. A constraint about the number of levels in Chatres is clearly more specific than a rule about the number of levels in Gothic cathedrals. Determining if one constraint is more specific than the other requires determining if the set of possible constants the first constraint applies to is a subset of the set of possible constants the second constraint applies to, if they both apply to the same set, or if the first constraint is a superset of the second. The following chapter explains the method used to determine if one constraint is more specific than the other.

The term rebut is used in the ARC Logic system to refer to conflicting constraints that apply to the same set, because the constraint defeats the old conclusions drawn. When one constraint applies to a subset of the other, this is referred to as undercutting in the ARC Logic system. The more-specific constraint defeats the reasoning of the over-generalizing constraint, and the over-generalizing constraint is modified appropriately. The exact approach of the ARC Logic's rebutting and undercutting of constraints, and how this differs from Pollock's use of the terms, is also explained in detail in the following chapter.

CHAPTER 6

IMPLEMENTATION OF DEFEASIBILITY

In classical logic and traditional Prolog, proven consequents can be separated from their consequence relations. The set $\{P,\ P \rightarrow Q\} \models Q$, so Q can then be used without consideration of the set that entailed it. In a defeasible system however, the consequents of rules may "not be detachable even when their antecedents are derivable," as one of these detached consequents could be defeated by additional information [10]. In the Tweety triangle example, the reasons to believe that Tweety flies and the reasons to believe that Tweety does not fly are necessary to determine how the conflict should be resolved. Without this information, there is no information that distinguishes the justification for the two conclusions.

The ARC Logic system has defeasibility functionality as a result of three components. First, ARC's logic system has a way to hold meta-information about facts and rules. It is not enough to say some *P* is true or false; the system needs to know the certainty of the truth value of *P*. Secondly, the system has a defeasibility-preserving method of inference that uses this meta-information of premises to determine the defeasibility of conclusions. Thirdly, the system has a method for comparing conflicting lines of reasoning. This chapter explains how these three components are implemented in the ARC Logic system, and briefly discusses how they differ from other conceptions of defeasible reasoning in order to fit the domain and demands of this implementation.

6.1 DEFEASIBLE KNOWLEDGE REPRESENTATION

Chapter 4 discussed the knowledge representation in the ARC Logic system, but intentionally ignored the aspects of the representation that are only necessary for defeasibility functionality. This section augments chapter 4 by explaining how these functions are built into all three types of domain knowledge: facts, constraints, and term definitions.

6.1.1 Defeasibility of Facts

In defeasible reasoning, consequents may not be detachable from their consequence relations, which is problematic for a forward-chaining inference system like the ARC Logic system. A forward-chaining system makes all derivable information explicit, instead of waiting for a goal to match. In a backward-chaining program, a goal (query) must be determined first, then the system works backwards to prove all the goals necessary to prove that first goal. Prolog is a backward-chaining system, requiring a goal up front, and technically, any program written in Prolog requires backward chaining to solve a goal or goals. The ARC Logic inference engine is itself essentially just a query. The ARC Logic system can be said to be a forward-chaining system because it fills the knowledge base with all the information that can be inferred, in the form of facts. To do so requires detaching the derivable consequents from their antecedents, but a complete detachment is not possible in a defeasible reasoning system [10]. This implementation circumvents the need to keep facts tied to their method of proof, by holding two important pieces of metadata for each fact in the knowledge base.

The first piece of metadata is the defeasibility of the fact, which can either be defeasible or indefeasible. The input method for asserting facts is assert_fact(+X, +Defeasibility, -Asserted). If the user writes an assert fact input with only one argument, such as

assert_fact(above(2,3)), then the ARC Logic system defaults to indefeasible. The difference between defeasible and indefeasible should be clear, and the way these two are compared will be clear later in this chapter.

The other piece of metadata, the origin, is necessary to allow for removal of defeasible facts that should no longer be assumed. Because the system is non-monotonic, facts in the knowledge base are not necessarily "safe" from new information. If some fact or rule is modified, removed from, or added to the knowledge base, the system needs a way to expunge all the facts that were consequents of the no-longer-usable information, or those facts that were consequents of the lack of some information that is now added. Assume the KB contains {*P, R, P* → *Q, (R unless S)* → *T*}. From these, *Q* and *T* are inferable. If P is removed from the KB, then *Q* no longer follows, and it is important that the ARC Logic system recognizes this. The same is true for *T* if *R* is removed. The *(R unless S)* → *T* implication also means that if some new information *S* is added, then *T* does not follow. In the ARC Logic system's implementation of defeasible reasoning, information that is a consequent of any defeasible fact or rule is itself defeasible. Since only defeasible rules and facts can be modified or removed from the knowledge base, all the information in the knowledge base that could potentially become inderivable is also defeasible.

When the inference engine is called, before it forward-chains to finds everything derivable from the current knowledge base, it clears away anything that could possibly be incorrect in light of the new information by retracting all facts that are defeasible. After the retractall is used, the KB will only contain facts that have justification, but will not necessarily contain all the facts that have justification. Even after inference takes place, there is still the possibility that not all facts that are justified are in the KB. Facts can be justified based on their

derivability (defeasible or classic entailment), but they are all justified if they are entered by the user directly.

In a classical, monotonic, logic system, if *P* is in the knowledge base, then the system treats *P* as true. The logical inference makes no claims to the truth of *P*, only what can be derivable assuming *P* is true. In the ARC Logic system the user can directly input a defeasible fact, which has the justification to be assumed true unless there is some reason not to. If the only condition for a fact falling under the axe of retractall is that it is defeasible, it will also remove user asserted defeasible facts. Unlike inferred defeasible facts, they will not return when the inference engine is run. The function to retract all the facts must be slightly more precise to ensure that every fact that is justified, whether from inference or because it was added directly, is in the knowledge base. The second piece of metadata stored with each fact is the origin of the fact, which can be either inferred or explicit. When the assert fact input method is called, it checks the value of the origin flag and uses this value to as the second piece of metadata, which is the third argument, for fact/3. The origin flag is set to explicit by default, and changed to inferred temporarily whenever the inference engine is run, since this inference process uses the same assert fact input methods. Instead of retracting all defeasible facts at the beginning of a call to the inference engine, the ARC Logic system retracts only those facts that are defeasible AND inferred.

The defeasible parts of the ARC Logic system's knowledge representation are rather generally straightforward. One point of complexity arises in the use of objects. Object facts are designed, for simplicity, to be a hybrid of one fact about type and one about existence. The certainty of the type of an object is generally an irrelevant concern in this domain. It is unlikely that a user would want to create an object instance for some object without knowing what the

object is. It is very possible that one would want to create constraints for uncertain types of objects, such as "each arch is above two supports," without needing to say that these objects are columns or piers. The current implementation of the ARC Logic system does not allow for this. For balance between simplicity and versatility, constraints are designed so that a simple type is used as the last clause. Supertypes are only to be used in the condition clause of constraints and queries. For the sake of simplicity, assumptions were made in the ARC Logic system that each object instance has a type, that this type is certain, and that there is no reason to have information about the type of some instance if that instance does not exist. In this system, the certainty of the type is irrelevant, but the certainty of the existence of the object instance is crucial, and it is this value that is reflected in the defeasibility metadata for an object fact.

6.1.2 Defeasibility of Constraints

In the use of the ARC Logic system, most facts will be inferred from constraints and term definitions entered by the user. Both of these types of domain information allow the user to express aspects of defeasibility. These indications of defeasibility are used when the constraints and logical properties of terms are converted into rules, and are used by d_call to ensure the facts derived from these rules are asserted with the correct defeasibility.

When create_constraint was explained in a previous chapter, the third argument was ignored. This argument represents the defeasibility of the constraint. When the value of the third argument is must, the constraint is considered indefeasible. When the value is d_must, generally, or presumably, the constraint is considered defeasible. The defeasibility of a constraint matters for two related but separable reasons. First, it matters in the way conflicting constraints are compared, as discussed later in this chapter. Secondly, the defeasibility of the

constraint is a factor in the determination of the defeasibility of the consequents from the constraint. The third line in the body of a rule made from create_constraint will say either defeasible or indefeasible, which, when called by d_call, will always be defeasibly true or indefeasibly true, respectively.

6.1.3 Defeasibility of Term Definitions

In the ARC Logic system, terms are considered global and monotonic. The assumption is that a term has the same meaning for the duration of a session and in all contexts within a session. Term definitions therefore are indefeasible; they cannot be contradicted or excepted. If above is defined as transitive, asymmetric, and irreflexive, the assumption made by this implementation is that above should hold this behavior throughout. Aboveness has the same implied behavior when talking about parts of a column or the vaults and the floor.

The concept that terms, both relationship terms and object terms, have one meaning in one context and a separate meaning in another context is understandable, but implementing this would likely add more confusion than functionality. If a user wants to define terminology that works differently in different contexts, this can be done by making an adjustment in taxonomy rather than by adding defeasibility to terminology. Term names are defined by the user, so if a term has a different semantic value in two different contexts, a naming convention can be devised to separate these into two or more distinct concepts with different names, similar to the tutorial's use of above and immediately_above.

While term definitions themselves contain no defeasibility functionality, individual logical properties of terms do. Unlike the create_constraint method, which creates only one new Prolog rule for the KB, the define_relationship method creates one or more rules for each

logical property in the list. It is the individual logical properties, and the Prolog rules that follow from them, which can be defeasible or indefeasible reasons. While defeasible logical property rules cannot be modified like defeasible constraints, they both allow for the user to indicate the defeasibility of the reasoning itself. The six logical properties already listed for define_relationship/2 in chapter 4 are indefeasible. The transitivity of above, for example, would be indefeasibly true in almost any possible usage. There may be some logical properties of relations however that would be useful to assume, but that are not certainties. The user can create relationships with defeasible variations of any of the positive or negative logical properties, by using the same property atom but starting with a "d_." For example, a user may want to represent a relationship for 'bears', as in 'bears some of the weight of'.' If *a* bears some of the weight of *b*, *b* bears some of the weight of *c*, the user may want the system to automatically infer that *a* defeasibly bears some of the weight of *c*. This can be done by entering define_relationship(bears, [d_transitive]). The defeasibility aspect of the creation of Prolog rules from logical properties works the exact same way as with constraints; either an always-defeasible or always-indefeasible clause is added to the body of the rule. The same principle and method works for define_metarelationship. For example, the user could include define_metarelationship(d_implies, below, bears), since it might be useful to assume, but not certainly true, that if an object is below another object it is bearing the weight of that object.

6.2 DEFEASIBILITY-PRESERVING INFERENCE

The previous section explained the way metadata holds information about facts, and the way defeasibility is included in the rules from constraints and the logical properties of term definitions. To use this information to prove goals, while also tracking the certainty of the proof,

requires a valid method of reasoning. In classical logic, a method of reasoning is valid if it is truth preserving. The reasoning method of the ARC Logic system must be both truth-preserving and certainty-of-truth-preserving.

Prolog proves some_goal by backward-chaining to prove any other goal that could lead to the proof of some_goal. When this inference system of Prolog needs to be called explicitly, it is done with the call/1 predicate, such as call(some_goal). The ARC Logic extends the inference capability of Prolog with the predicate d_call/2. When the user runs the query q(object(X,Y)), they are making a call to the d_call predicate, and when the inference engine finds everything derivable it also uses the d_call function.

It is because of the work of the d_call function that the rules created from constraints and logical properties of terms look so simple. These rules do not need to extract facts from their encapsulation in metadata, or concern themselves with assigning the right defeasibility to the head of the rule; this is all done by the d_call function. It is important to note that the d_call predicate does not replace the inference system built in to Prolog, nor does it break the rules of this system. The d_call predicate itself is called by the standard Prolog inference. Defeasibility is added to Prolog in a manner similar to the way explicit negation is added. Prolog considers not_above(1,2) to be true, which is only considered by the users and the ARC Logic extension as equivalent to above(1,2) being explicitly false. In the same way, Prolog inference considers fact(above(1,2), defeasible, inferred) in the knowledge base to be simply true. As far as the Prolog system itself is concerned, it is certainly true that (above(1,2) is uncertainly true). The extension of the explicit negation and defeasibility in the ARC Logic system is a maneuver to make Prolog behave as though it had explicit negation and defeasibility, without changing the way Prolog itself works.

Since the d_call predicate is only a slight extension of the way the standard call predicate works, it is rather simple to implement and understand. In standard Prolog, a fact such as bird(tweety) is just a shorthand version of bird(tweety) :- true. To prove flies(tweety), the Prolog inference works its way through the knowledge base, unifying with heads of rules and taking each clause in the body as a new goal to be proven. flies(tweety) is matched with the head of flies(X) :- bird(X), so the new goal becomes bird(tweety). If the goal (or each goal in a conjunction) leads to a true, then flies(tweety) can be validly inferred.

d_call works the same way standard Prolog inference does, but keeps track of one additional piece of information as it goes. If d_call succeeds, it unifies the defeasibility of the success with a variable in the second argument. When d_call uses an ARC fact to prove a goal, it takes the defeasibility metadata about that fact as the certainty of the reason to support the goal. Of course the d_call predicate also uses Prolog rules, created from constraints and logical properties of terms, in order to prove goals. If above is in the ontology and is defined as transitive, then the following rule is in the knowledge base:

```
above(A, C) :-
        d_ground(above(A, B)),
        above(B, C),
        indefeasible.
```

To prove above(A,C), each clause in the body becomes a goal that has to be proven, just like in a regular Prolog call. Instead of simply determining if a clause can be proven, d_call determines if a goal can be proven and the defeasibility of that proof. When a conjunction of clauses is needed to prove a goal, the goal is given the same defeasibility as the weakest link; if any one of the clauses is only provable defeasibly, the new fact must also be defeasible. The d_call function uses a predicate that returns all the alternatives from back-tracking, just like

pushing the semicolon on a query to see all possible matches. If through backtracking, alternative proof methods are found, then the strongest defeasibility amongst these is assigned to the new information, because if a fact can be proven through a defeasible chain of reasoning and an indefeasible chain of reasoning, then it is known indefeasibly. This is also true for queries and constraint conditions that include disjunction directly with a semicolon.

In the ARC Logic implementation, indefeasible vs. defeasible of d_call/2 works theoretically the same as true vs. false in classical logic. One false/defeasible clause makes a conjunction false/defeasible, and one true/indefeasible clause makes a disjunction true/indefeasible. In the ARC Logic implementation, no matter how much defeasible information is required to prove a goal, the goal is considered defeasibly true.

There are special cases where the d_call predicate does not function as stated above. The clause that simply says indefeasible (left out of the examples in chapter 4) always succeeds with the defeasibility of indefeasible. Of course, if the clause were defeasible, it always succeeds defeasibly. With this clause, the defeasibility of the logical property itself can play its part in determining the defeasibility of above(A,C). If above was defined with d_transitive, then the proof of above(A,C) would be always be defeasible, even if above(A,B) and above(B,C) can both be proven indefeasibly. Defeasible rules from constraints and logical properties can only lead to defeasible facts, while an indefeasible rule could prove facts as indefeasibly true, as long as all the other clauses of the body can be proven indefeasibly true.

The d_call predicate is designed to work only with the domain data itself, which is in the form of pure Horn clauses. Prolog has procedural predicates that cannot be written as pure Horn clauses. When the d_call encounters predicates like print(X) or =< (less than or equal to), the d_call predicate just calls these clauses with the standard call predicate. An important procedural

function in Prolog is the cut (!), which prevents back-tracking. While none of the rules from domain information use a cut directly, they may call predicates that require using cuts, so d_call calls these methods with standard Prolog inference, and the methods explicitly include code that performs the duties of assessing defeasibility that are generally left to d_call. The cut is needed for the negation-by-failure unless clause. Because negation-by-failure is defeasibly false, when unless(X) succeeds (because X fails), it does so only defeasibly. The only exception to this is when the unless predicate is used to call min_is_met, which makes no claim of defeasibility.

6.3 CONFLICT RESOLUTION

This thesis has so far explained how users enter defeasible domain information with input methods, how this information is translated into Prolog rules and facts, and how the defeasibility-preserving d_call predicate works. Together these allow for all the information in the knowledge base to be labeled correctly as defeasible or indefeasible. This thesis has yet to demonstrate, however, the use of this defeasibility information. The ARC Logic system is non-monotonic and defeasibility-preserving so that it can appropriately handle conflicting information. This section explains how conflicts, in facts and constraints, are resolved.

The ARC Logic inference engine queries the KB for all the facts it can, and then calls assert_fact (or assert_object_instance which ends up calling assert_fact for two separate facts). This is the same method the user can use to enter facts about relations and objects directly. Calling assert_fact(X) does not guarantee that X will be asserted to the knowledge base. When assert_fact is called, the candidate fact is compared to the information in the KB to check for conflicts, and handle the conflicts if there are any. The first subsection explains how fact comparison and conflict resolution work. When the user writes constraints with

create_constraint, these are also compared to the KB. The check and conflict resolution of constraints is explained in the second subsection of this section.

6.3.1 Fact Comparison

Whenever assert_fact(some_fact) is called, the knowledge base is checked for matching and conflicting facts. If some_fact is already in the knowledge base, then the metadata of the matching facts are compared. If the new version of the fact is indefeasible and the old one is defeasible, the old one is retracted and the new version of the fact is asserted. Just like with d_call, the strongest proof of a disjunction is used. If there are any defeasible facts that were derived using the replaced fact, these facts will also have the opportunity to be replaced with indefeasible versions, because the inference loop is run until it performs a complete cycle without asserting any new facts to the knowledge base.

In addition to checking the defeasibility, the origin is also checked for matching facts. If the origin of the new version is explicit, and the origin of the old version is inferred, then the new version replaces the old. The origin only matters when a fact is defeasible, and it is important that the strongest (the stronger being explicitly asserted) origin is used. If a defeasible fact *P* is only known through inference, then *P* is removed from the KB if there is no longer the ability to derive *P*. However, if *P* is explicitly stated it is not removed on this condition. A defeasible explicitly stated fact can only be removed by an explicit conflict.

The advantage of defeasibility is that it can (sometimes) handle *P* and ~*P*, depending on the justifications for each. When new fact assertions are checked, they are also compared to any fact they conflict with. Facts can conflict because one is the negated version of the other, such as above(1,2) and not_above(1,2), or because they are antonyms, such as above(1,2) and

below(1,2). If, in the process of assert_fact, a conflict is found, there are four possible outcomes of resolution/irresolution.

The following chart shows these four simply:

Newly Asserted / Already in KB	defeasible	indefeasible
defeasible	Both facts removed	Old fact removed New fact asserted
indefeasible	New fact ignored	Requires manual fix

Table 1. Defeasible Fact Comparison.

If the new fact being asserted is defeasible and the conflicting fact in the knowledge base is indefeasible, the new fact is ignored. If the new fact being asserted is indefeasible and the one in the knowledge base is defeasible, then the new fact is asserted to the knowledge base, and the defeasible fact is retracted. If both of the conflicting facts are defeasible, then the best resolution is to assume neither one to be the case, so the fact already in the knowledge base is retracted and the new one is ignored. If the conflicting facts are both indefeasible, this is the same problem that arises with contradictions in classical logic. ARC's logic system cannot automatically resolve this problem, and instead warns the user, because an indefeasible contradiction signals that either the user's ontology is self-inconsistent, or is constructed in a way that does not meet the specifications of the program.

6.3.2 Constraint Comparison

When create_constraint is called, the ARC Logic system compares it against all the constraints already in the KB in order to check for conflicting constraints. In fact assertion, redundant facts are not allowed; only one version, the strongest version, of each fact ended up in the knowledge base. Constraints that cause the inference engine to find redundant facts are not problematic, as they ultimately result in fact assertions anyways, where redundant facts are taken care of. The user can have a constraint say "all columns defeasibly have a necking" and then say "all columns in the clerestory certainly have a necking." Here, both constraints would apply to some column that is in the clerestory, but the indefeasible version will override the defeasible version when they are asserted to the knowledge base. The only time such redundancy is problematic is with object_to_assert facts. After the inference engine is finished querying, it has gathered all the facts to be asserted, and at this step it checks that there are not two or more identical object_to_assert facts, and only uses the strongest defeasibility if there is more than one.

Unlike fact assertions, which are checked for both matches and conflicts, constraints only have to be checked for conflicts. The ARC Logic system can resolve conflicting facts as long as at least one of the facts is defeasible; the same is true for constraints, but the procedure for handling conflicting constraints is more complex. There are two different ways in which constraints can conflict, and the terminology of rebutting defeaters and undercutting defeaters is borrowed from Pollock [9] to describe these two types of conflict in the ARC Logic system. With this approach there is no need for the user to indicate a constraint is undercutting or rebutting; the program automatically discovers if either is the case and deals with the constraints appropriately.

For constraints to conflict in either way, they must have the same object type for the consequent, and the minimum of one constraint must be greater than the maximum of the other or the relations must be explicit contradictions (either negation or antonym). This check is the only time that the maximum value is used in the ARC Logic system. The minimum could be considered a rule about positives, "at least this many need to exist," and the maximum is a negative rule "no more than this number can exist." This is why the inference engine is only concerned with ensuring the minimum is met.

For constraints to be in conflict, they also need to potentially be applicable to the same instances. The comparison between the condition clauses of the constraints determines whether conflicting constraints are considered rebutting or undercutting. When the condition is exactly the same, the constraints rebut each other. To check for exact sameness, which is used also for matching constraints, `=@=/2` is used instead of the standard unification. The use of variables in constraint conditions should not be considered the same as using specific atoms. For example `(object(column, X))` should not be considered equal to `(object(column, ex_column))`, but they are unified with the standard unification, so `=@=` is used instead of standard unification to check for matching constraint conditions. Rebutting constraints are dealt with in the same way as conflicting facts; as long as at least one is defeasible it can be resolved.

The ability to assume one thing, and then claim just the opposite, as is done with rebutting constraints, can be useful, but in order to match the style of natural descriptions, which often overgeneralizes and then makes exceptions, the ability to create generalizing constraints and exception constraints is necessary. If one constraint applies to a set *A*, where *A* includes every instance that meets some condition, and a conflicting constraint applies to only a subset of *A*, then this second constraint undercuts the first.

Returning to the column example, assume constraint(X, object(column, X),presumably, has, 1, 1, base) is already in the KB, and create_constraint(X, (object(column, X), contains(Y,X), object(arcade, Y)), presumably, has, 0, 0, base) is called. These two constraints translate to "it should be assumed that each column has a base" and "columns in the arcade level do not have a base." The latter constraint is clearly more specific than the former, and since it is also in conflict, the latter constraint undercuts the former. This undercutting is only possible if the constraint being undercut is defeasible. The defeasibility of the constraint doing the undercutting does not matter, but of course an undercutting defeasible constraint can itself be undercut.

As explained in the previous chapter, the ARC Logic system works with the natural assumption that the conclusion of a more specific rule should override a conflicting conclusion from a more general rule. Because of the particular nature of Gothic cathedrals and similar domains, specificity can be determined with a simpler process than it can be in the general system of d-Prolog [10]. If one condition undercuts another, every possible instance that can be matched to the more specific condition can also be matched to the general condition.

To programmatically determine if (object(column, X), contains(Y,X), object(arcade, Y)), (now on referred to as condition A), is more specific than object(column, X) (condition B), the ARC Logic system goes through each clause in condition B, and checks if it is "in" the condition for constraint A. This is a straightforward process with just a couple of exceptions. The basic functionality is similar to the built-in predicate member(?Elem,?List), which is true if Elem is in the List, but there is one important difference. member uses unification, so it can match a variable to an atom. When determining if one clause of a condition is inside another condition, the unification of a variable to an atom is only acceptable if the variable is in the

general condition and the ground version is in the specific condition. Clearly object(column, X) should be considered more general than object(column, my _column), but not the other way around, and the ARC Logic system is careful in its matching.

While this matching is used for most clauses, there are two special exceptions that are considered. The first exception for consideration is the use of subtypes. The condition object(column, X) is more specific than the object(support, X), assuming that the subtype relationship is defined. The other special kind of clause is the contains clause. The importance of determining specificity of contains clauses will be clear in the next chapter when scope is introduced. To know if contains(2, X) is more specific than contains(1, X), the predicate which checks for sub-conditions checks if contains(1, 2). This approach to determining specificity requires the use of additional information not present in the two clauses being checked, similar to the way d-Prolog solves the Tweety triangle [10].

The function which checks for sub-conditions also returns the difference between the two conditions. When conditions A and B are entered, the difference is ((contains, Y, X), object(arcade, Y)). This difference is used to resolve undercutting constraints. The more specific constraint is added to the knowledge base without modification. The more-general constraint is retracted and modified before it is asserted back to the knowledge base. The modification is an addition of the difference contained inside an unless clause. After this modification, condition B looks like: (object(column, X), unless((contains, Y, X), object(arcade, Y))).

After the comparison, modification, and assertions, the knowledge base from this example contains two constraints (in their Prolog rules format) corresponding to constraint(X, (object(column, X), contains(Y,X), object(arcade, Y)), presumably, has, 0, 0, base) and

constraint(X, (object(column, X), unless((contains(Y,X), object(arcade, Y)))), presumably, has, 1, 1, base). If some fact for object(column, ex_column) is in the KB, and d_call cannot prove (defeasibly or indefeasibly) that ex_column is contained in an arcade object, then a base is created that belongs to ex_column. Every time the inference engine is run, it starts by retracting all the facts that are defeasible and inferred. Each time, the facts for the base object and has relation, which are defeasible and inferred, are retracted, then added back because of the constraint. If some information is added which shows with certainty, or provides for the assumption, that ex_column is contained in an arcade object, then upon inference the base and has facts will be removed, and will not be asserted back by the inference engine as they are no longer derivable.

Consider what happens if create_constraint(X, (object(column, X), contains(Y,X), object(arcade, Y)), presumably, has, 2, 2, base) is called. This constraint rebuts the constraint already in the knowledge base. If a defeasible constraint is rebutted by another defeasible constraint, then neither constraint is used, so the only constraint remaining in the knowledge base is constraint(X, (object(column, X), unless((contains(Y,X), object(arcade, Y)))), presumably, has, 1, 1, base). If there is a column object, then a base is created for it, unless that column is in an arcade, in which case, there is no justification to assume anything about the column.

The user can also write constraints with unless clauses in the condition, and the undercutting works correctly. The ARC Logic system allows users to create constraints without explicitly including all possible exceptions, but the user can add these anyways. This might be useful if an exception is known, but the constraint applying to the exception is unknown. The user could write a constraint that applies to every Gothic cathedral except Chartres. The general

condition object(cathedral, X) is undercut by the more specific object(cathedral, X), unless(object(cathedral, chartres)). When the first condition is modified, it becomes object(cathedral, X), unless(unless(object(cathedral, chartres))). An unless clause inside an unless clause works the way one would expect a double negation to work, except no values are matched to variables on success. The double negation is automatically changed by the program to a positive version so that future comparisons do not need to take the double negation equivalency into consideration. Therefore, the modified condition ends up as object(cathedral, X), object(cathedral, chartres) and it will only be applicable to Chartres cathedral.

The primary reason for using defeasibility to model natural language is the ability to write general and exceptional rules, in a way that does not require explicitly making all exceptions. The ARC Logic system gives users this ability, while ensuring that the KB contains no overgeneralizations. All the exceptions the user would need to make explicitly in a monotonic language are automatically added explicitly by the system itself, to ensure that no constant can match the conditions for contradictory constraints. This is one more way the ARC Logic system extends Prolog by converting defeasible logic concepts into facts and rules that can be handled by Prolog's internal system.

The method for defeasible knowledge representation, inference, and the use of defeaters is adapted from formalized descriptions of defeasibility to fit the ARC Logic system's particular implementation. In this implementation, the facts themselves hold a measure of defeasibility, allowing them to be separated from the consequence relations they came from, and allowing users to directly assert defeasible facts in order to work with uncertain information. A valid defeasibility-preserving inference in this implementation holds, no matter how much defeasible information is used in order to prove the goal. Finally, the terminology for defeasible reasons is

adapted from their formalized descriptions for this implementation. The ARC Logic system's defeasible constraints are equivalent to 'prima facie reasons' [9], or 'backing clauses' [12] in other conceptions of defeasible reasoning. For example, $\{P, P\ (d\rightarrow)\ Q\} \models dQ$, such that the backing clause combined with proof for the antecedent entails the defeasible assumption of the consequent.

The concept of defeaters is used in the ARC Logic system but this concept is adapted for this domain and the goals of ARC Logic implementation. Defeaters only provide a reason against the use of some defeasible reasons. Defeaters do not allow anything new to be derived; they only provide a reason to avoid making a bad assumption. In the ARC Logic system, there are not explicit defeaters. Because explicit negation is added to the system as a work-around of Prolog's internal inference, explicitly-false facts are treated by Prolog the same as explicitly-true facts. In the same way, the ARC Logic system does not have explicit defeaters, only constraints that can conflict with other constraints. Even these negative constraints vary from the idea of defeaters because they can potentially be a reason to add new facts to the knowledge base.

Because defeaters are implemented differently in the ARC Logic system, the terminology for undercutting and rebutting defeaters is also adapted from Pollock's version [9]. Rebutting defeaters are those that attack a conclusion that comes from defeasible reasons [9]. In the ARC Logic system, if one constraint says "capitals are defeasibly above bases" and another constraint says "capitals are defeasibly not above bases," the constraints rebut each other. They draw opposite conclusions and together provide a reason to not assume either conclusion. If one of these rebutting constraints was indefeasible, it would remain in the knowledge base, and could derive new facts.

Pollock distinguishes rebutting defeaters from undercutting defeaters, which attack the reasoning that lead to the conclusion, instead of attacking the conclusion itself [9]. In the ARC Logic system, constraints are only undercut because they overgeneralize; the constraint makes assumptions about entire sets when the constraint cannot be applied to the entire set. If the constraint "all columns defeasibly have a base" (constraint A) is in the KB and a new constraint is added saying "all columns in the clerestory defeasibly do not have a base" (constraint B), then constraint B undercuts constraint A. In the ARC Logic system, the more-specific constraint does not give a justification for no longer using the general constraint. If object(column, 1) is in the KB and there are no facts saying it is contained in a clerestory, then object(base, 2) and has(1,2) will be added to the KB because of constraint A. Constraint B does not provide any reason to not assume object(base, 2) and has(1,2), it only provides a reason to deny the overgeneralizing reason of constraint A.

Constraint undercutting could be considered a two-step process in theory. The first step is in line with Pollock's conception of undercutting defeaters, because the undercutting constraint B attacks the reasoning behind constraint A. Constraint B does so by providing justification for believing that constraint A is overreaching in its implication. The implication about bases in clerestory columns is wrongfully lumped in with an implication about bases for columns in general. To resolve this overreach, constraint A is split, along the known fault line, into "all columns in the clerestory defeasibly have a base" (constraint A1) and "all columns unless they are in the clerestory have a base" (constraint A2). The second part of constraint undercutting is close to rebutting, because constraint B provides a reason against constraint A1. Even though constraint B is defeasible (indefeasible constraints can be undercutters also), it has justification to be used because it was explicitly created, unlike A1, and because of the priority of specificity.

The ARC Logic system's implementation of defeasibility allows the user to assert uncertain facts and uncertain constraints. It also allows the user to write overgeneralizing constraints and then write constraints that are exceptions to other constraints without any need to indicate or even notice when this is being done. These non-monotonic aspects bring the ARC Logic system a step closer to natural description than is possible with monotonic logic and conventional Prolog.

CHAPTER 7

SCOPE AND THE USE OF DEFAULT DESCRIPTIONS

Throughout this thesis, the majority of examples have been limited to columns. The column case study allowed for the explanation of the knowledge representation, input methods, fact and constraint comparison, the inference engine, and querying the knowledge base. Now that all of these components have been explained, they can be combined, and the purview can be expanded, with a brief introduction to the description of an entire Gothic cathedral. This chapter demonstrates how all the components of the ARC Logic system already discussed can be scaled up to the level of entire cathedrals and beyond, and introduces the scope method which facilitates the use of the ARC Logic system on a large scale.

7.1 DESCRIBING DEFAULT AND SPECIFIC CATHEDRALS

The ARC Logic system is developed to be flexible, and it is adaptable to serve different uses, different ontologies, and even different domains. The choices made in the example description of a column may have seemed like the obvious choices, but even in something as small as a column there is possible variation of description. Depending on what aspects of the description a user is focused on, their taxonomy, and the default assumptions they choose to work with, methods for describing the same scene can still vary significantly. This is true with columns, but it is especially true with entire Gothic cathedrals. This chapter will provide the start

of a description of a cathedral simply to highlight the methods available to the user, not to designate a standardized way in which Gothic cathedrals must be described.

The number of stories in the nave of a Gothic cathedral is an important part of an architectural description, but this number varies from cathedral to cathedral. A description of a generic, default, Gothic cathedral could say that a cathedral indefeasibly has a nave, and that a nave indefeasibly has an arcade and clerestory, and defeasibly has a gallery and triforium, because these levels are not necessarily present in every Gothic cathedral. The vertical sections created split by columns that run along the side of the nave are called bays, and these can also vary in number depending on the cathedral. The default cathedral could say that there are indefeasibly a minimum of 4 and a maximum of 10 bays along the side, because it is known that the number of bays in every Gothic cathedral falls within this range, or because if an object falls outside this range it should not be classified as a Gothic cathedral. Even though there has to be at least 4 bays, the generic model could say that the nave defeasibly has exactly 8 bays running along the side. The ARC inference engine will create 4 indefeasible bays and an additional 4 defeasible ones.

If the user wants to describe a specific cathedral, they can load all of these constraints written for the default cathedral into the knowledge base, and then write constraints for those things where the specific Gothic cathedral varies from the default model. To describe the cathedral in Chartres, the user would assert an object instance of a cathedral named chartres. Then, any constraints that are specifically about Chartres, will contain in the condition argument, the clause contains(chartres, X), or whatever variable is used for the universal quantifier.

To describe the three-story cathedral in Chartres, the user would likely want to create a constraint, either defeasible or indefeasible, saying that Chartres has no gallery. The constraint

which said cathedrals defeasibly have a gallery is undercut by the more specific rule that pertains to just Chartres. The user can next enter a constraint that Chartres has 7 bays along the side of the nave. This constraint does not conflict with the indefeasible constraint that says there is a minimum of 4 and a maximum of 10 bays, but it does conflict with the defeasible constraint which says there are exactly 8 bays. Since the constraint pertains only to Chartres it is more specific so it undercuts this constraint it conflicts with. If inference has already been run after the Chartres cathedral object was asserted, there are defeasible facts for 4 bays in Chartres in the KB. These defeasible facts will be retracted when inference is run again, and instead 3 bays will be added to the KB because of the undercutting constraint about bays in Chartres.

7.2 SCOPE AND CONTAINERS

The constraints specifically about Chartres undercut the default constraints because they are more specific. The assumption is that one would add contains(chartres, X) to the conditions of each of the constraints. Manually including this information in each constraint about the specific cathedral is awkward and unintuitive. To correct this, the ARC logic system includes a way to alter the scope of the conversation. In natural description, the scope of the description is often explicitly stated or at least implied. It is often very clear if a description is referencing a concept in general or a specific instance that fits that concept. In the same way, a natural description may be implicitly limited to a particular section of a cathedral or particular object, and it is implied that the same rule cannot be applied everywhere throughout the cathedral. The scope can be considered a "zooming" function that allows one to designate that the following constraints describe only a particular area of a cathedral, just as they can designate that some constraints only apply to the cathedral in Chartres or Notre Dame de Paris.

The user accesses this ability with the predicate set_scope/1. The scope indicates the outermost container that is being described. Whenever a constraint is created, a contains(Scope, X) clause, referred to from here on as the scope clause, is added to the condition, where Scope is the current scope set in the knowledge base and X matches the universal quantifier (the first argument) of the constraint.

When describing a default model, the user can call set_scope(default), or any other name to give to the default model, then create constraints, which will automatically include contains(default, X) in their conditions. In this case, default is not the name of any object, but is rather just the name of a container. When object facts were described in an earlier chapter, it was explained that all objects must belong to some other object or a container, and that the very top-level (which is 0 by default) could not be an object because then it would require something to belong to. A container designates a level at which constraints hold, and each object instance that is contained in the container inherits the properties of these constraints, though these constraints may be undercut.

To use the default model, the user needs to assert a cathedral object, either named default, or named anything but have default as the value for the IsPartOf argument. The contains relationship is reflexive, so an object contains itself. If the user creates a cathedral called default, any constraints that include contains(default, X) in the conditions will be applicable.

To describe Chartres Cathedral, the user would write assert_object_instance(cathedral, default, chartres). This indicates that the specific cathedral of chartres is contained in the container default, so chartres will inherit all the rules that apply to the default cathedral. Then the user can set_scope(chartres) and all the constraints written from this point forward (until the scope is changed) will only be applicable to

chartres. Any constraints that conflict with the default constraints will undercut those constraints because the constraints about Chartres are more specific.

7.3 DEFAULT DESCRIPTION HIERARCHIES

The notion that an implementation of the ARC project's goals would require some way to fill in background information, and that this would require the use of a default cathedral, came early in the research of the project. The original conception for the default cathedral was quite different however. The original conception was to have a single default description file, which would be used to fill in the gaps wherever they result from a user's description of a specific cathedral. The work of this thesis started under the assumption that the logic engine would require melding a default description with a user description, but it became clear that the non-monotonic power was being wasted with this approach. In a description of a Gothic cathedral, there is not just a split between the general cathedral and a specific cathedral; there are splits between the general and the specific throughout. If a user can make assumptions about a generic cathedral object, which can be excepted appropriately, there is no reason why the user should not be able to do the same with a generic column object, or a generic nave, or clerestory, or base. Because of familiarity with Object Oriented concepts, the idea of the default cathedral being the cathedral class, and any specific cathedral, like Chartres, being an instance of that class, was difficult to overcome. Instead, Chartres Cathedral and a default Gothic cathedral are just like any other objects in the ARC Logic system.

The breaking of the class/instance model to use a flexible multi-level approach opens up many possibilities for variation in user description. Just as one can use generalization and exception with the objects inside the Gothic cathedral, constraints can be written at levels much

more general than Gothic cathedrals. The user for instance, could set_scope(generic_bulding), and then create a description for a generic building with constraints such as "all buildings have a floor," "all buildings have a ceiling," "the ceiling must be above the floor," etc. The user can then set_scope(generic_gothic_cathedral). By using assert_fact(has(generic_building, generic_gothic_cathedral), the user designates that the container generic_gothic_cathedral is inside the container generic_building, so all constraints about generic_building will apply to objects contained in generic_gothic_cathedral. If the user adds define_object(subtype, cathedral, building), then all the constraints that apply to buildings will apply to cathedral objects, so it will be inferred that each cathedral object has a floor and ceiling. Additional constraints can be written at this generic_gothic_cathedral level, but only those aspects of a Gothic cathedral that vary from a generic building need to be written exclusively. With this description of a generic Gothic cathedral, the user could write a description for a specific cathedral like Chartres, which will inherit constraints from the generic Gothic cathedral level and the generic building level. The concept of a class/instance dichotomy is no longer present, but the concept of class hierarchy and inheritance is. Just as classes can have multiple subclasses, higher-level descriptions can be used by separate, more-specific, descriptions. A user could create a generic building object, a generic Gothic cathedral object, a Chartres Cathedral object, and a Notre Dame de Paris object, all in a single session.

The ability to have many levels of hierarchy of description, branching sub-containers for description, and the ability to create any number of instances of different objects at different levels allows the user the freedom for comparing logical models of just about anything. The user could compare two or more different cathedrals, or compare a cathedral(s) against the default cathedral. The user could also compare different versions of the same cathedral over time, as

many have been modified extensively for various reasons. If the user is working with a description of Chartres from 1312 for example, they could set the scope to chartres_1312, and add constraints, then set the scope to chartres_1593 to add constraints from a description from that year. Then the user could create a cathedral object named chartres_1593 and one named chartres_1312, and run queries comparing the two.

The ARC Logic implementation allows for a high degree of flexibility in the method used for description and comparison of Gothic cathedrals. Constraints automatically undercut and rebut, so users need not concern themselves with conflicting constraints, and the scope function allows users the ease of shifting the context under which those constraints hold. The ARC Logic system's implementation eschews the use of a single default model for filling in background information of an incomplete description in favor of a more fluid method. The ARC Logic system can be used in a manner reflecting the original conception of a single default model, but the focus on a cathedral object is an arbitrary choice from the vantage of the ARC Logic system. A description of a default Gothic cathedral works the same way as a description about a column, or a description of the physical world in general, and all the necessary background information can be constructed within a hierarchy of inheriting descriptions.

CHAPTER 8
CONCLUSION

The ARC Logic system can be extended in a number of different ways. This final chapter will first discuss additional features that could be added to the ARC Logic system, and introduce possible modifications to improve the time complexity of the system. The ARC Logic system can be used as a standalone program, but the intention from the beginning is that it would be extended to include input components, like natural language processing of natural text descriptions, and output components, such as three-dimensional renderings of the logical structure of a described cathedral. The second section of this chapter will explain how these components, particularly the natural language processing component, will integrate with the ARC Logic system, and how the nature of the ARC Logic implementation facilitates accurate natural language processing. Finally, the third section will briefly highlight some possible domains outside of Gothic cathedrals that ARC Logic system can be used for with little or no modification.

8.1 EXTENSIONS OF THE ARC LOGIC IMPLEMENTATION

The goal of ARC Logic system is a useful and feature-rich implementation, which is not too complicated. To achieve this, decisions were made to favor some aspects of simplicity of use over greater expressive power. While this thesis touches on the justifications for these decisions,

they are certainly not the only approach; there are numerous ways the ARC Logic system's implementation can be modified or extended with other features.

One aspect of the ARC Logic system that required balance between simplicity and versatility is the use of constraints. Constraints allow users to write complex rules about the relationships and object without needing to understand Prolog, but the types of constraints that can be written are restricted. There are only two types of constraints: "any object matching some condition has a minimum and maximum number of object of a particular type" and "any object matching some condition shares a relation with a minimum and maximum number, or all, of sibling objects of some particular type." These two types of constraints have almost endless customizability and make up the majority of what would be described in a cathedral, but there could be a good reason for the ARC Logic system to allow expression of additional types of constraints. The use of constraints could be extended to allow for user-defined relationships between non-sibling objects, for example. Chapter 4 includes a brief justification for this limitation to sibling objects in the ARC Logic implementation, but it is possible for this to be extended for some particular purpose. One difference between `has` constraints and user-defined relationship constraints is that only `has` constraints create new objects. The constraint implementation could be extended to include a way to indicate that "each capital is above a shaft" is existence-enforcing, so that a shaft is created for each capital if it were not already present. With more expressive power of constraints, the `create_constraint` input method would likely also increase in complexity.

In Chapter 6, the inability to use supertypes in the consequent of a constraint, or assert them directly, was discussed. A constraint stating "each arch is above two supports" is not possible in the current implementation of the ARC Logic system. This is a not a result of the way

constraints work, but rather comes from the assumptions made about objects. If the fact about the type of an object were separated from the fact about existence of the object, then defeasibility of type could be implemented, and the system could work with objects while being uncertain of their type.

Chapter 6 also explained the reasoning for keeping relationship term definitions global. If there was some need for relationship terms to have different behaviors in different contexts, then the ARC Logic system could be altered to do this in a way almost identical to the rebutting and undercutting of constraints. Relationships in the ARC Logic system are always binary relationships, but the system could be extended to include adjectives, which are either unary relationships, or binary relationships between some object and a value. Adjectives have been avoided in the ARC Logic implementation for two reasons. First, it has not been determined that adjectives would add any useful expressiveness to the logical description of Gothic cathedral architecture. Second, there are numerous methods by which one can represent adjectives, and finding an optimal implementation would require some specification of the way they would be used in the domain.

One very dramatic addition to the ARC Logic system would be the ability to model the knowledge representation of multiple agents. The ARC Logic system implements defeasibility by moving outwards one meta-level, in order to not only talk about what is known, but what is known (the certainty) about what is known. While the ARC Logic system can be very useful for comparing different descriptions of the same cathedral, the ability to logically represent the descriptions (as agents) and their knowledge brings the logical description ability out another level. The user could model information like "every description of a cathedral (defeasibly) states that cathedrals (defeasibly) have 4 stories" and "John's description of a cathedral (indefeasibly)

states that cathedrals (defeasibly) have 3 stories." Some functionality along these lines could be very useful for an examination of the method of description itself, comparing the way description differs between people, periods of time, or even how description of Gothic cathedrals varies from descriptions of other domains.

Each of these possible extensions to the ARC Logic system would add some level of functionality, and also likely add some complexity for operating the system. There are some possible modifications to the ARC Logic system that do not change the functionality, but could improve the implementation. The concern of this implementation has been functionality over optimization of space or time, and there are likely some places for optimization. The exception of the contains relationship from the forward-chaining of facts was one implementation decision made partially for consideration of time. The component of the system that could potentially be sped up by a modification is of course the inference engine. The ARC Logic system's implementation of the inference engine is a looping system. On each loop through the system, the inference engine uses d_call, which is also used by the q for the user to query the KB. All facts that can be derived, even those that come directly from facts in the KB, are returned in a set, and then an assert_fact is attempted on each one of these. The time of the inference is linearly proportional to the number of loops through the inference engine that must be made before there are no new facts to assert. Some modification that would cut the search space, or the number of assert_fact calls, would increase the speed of inference.

8.2 USE AS COMPONENT IN THE ARC PROJECT

The examples throughout this thesis have used the ARC Logic system as a standalone program, but from the beginning the intention was to use the ARC Logic system as a component

of a larger implementation. To work in both these ways, the ARC Logic system was designed to be very modular. The ARC Logic system is designed be work as a black-box. The user needs to use the correct syntax for input methods and understand what the system does in a very general, theoretical sense, in order to use the system.

Output from the system is simply the listing (or query-answering) of the facts derivable from all the domain information that was entered into the system. One goal of the complete ARC project implementation is the inclusion of software which takes the logical fact output and creates two-dimensional or three-dimensional graphical renderings of the description. The ARC Logic system outputs facts in line with the original conception of a logical inference engine for the ARC project, and there is nothing particular to this implementation that would help or hinder the creation of visualization software or any other use of the output data.

Though the output of the ARC Logic system works the way it was originally conceived, the type and method of input is significantly different than the original conception. A major goal of the ARC project is the creation of a natural language processing system that automatically extracts all the relevant information, and turns it into input for the ARC Logic system. The knowledge representation and method of input is of major significance to the design of a natural language processing component, and how it would fit into the system.

The most basic way the ARC Logic implementation helps facilitates accurate input from the natural language processing component is by finding logical inconsistencies. Natural language processing is a difficult task because of the complexity and range of natural language, so incorrectly processed information is inevitable. Pointing out logical inconsistencies in the extracted information is one of the main benefits to performing inference on the information. If

at least one of the conflicting facts is defeasible, the ARC Logic system can handle the inconsistencies automatically, and avoid asserting facts that do not follow from the description.

In the original conception for the ARC project system flow, natural language descriptions are converted to a simpler and limited set of English, the domain-specific Architectural Description Language (ADL) [2] [3]. ADL allows the user to enter sentences like "A column is a type of support. Every column has a base, a shaft, and a capital. Most columns have a plinth. The base is above the plinth, the shaft is above the base, and the capital is above the shaft." [3] This simple subset of English would then be converted into Prolog rules and be asserted into the knowledge base.

The work of the translation from ADL to Prolog rules is achieved by the ARC Logic's input methods. The create_constraint method for instance, would enter the equivalent of the ADL proposition "Every column has a base" with create_constraint(column, must, has, base). ADL could potentially be an easier method of input, so the system could be extended to use this. The mapping from ADL to the ARC Logic system's input methods would be straightforward. The standardized arguments of input methods available in the ARC Logic system could be more useful than an ADL system, especially with other types of input, like a graphic user interface that would allow the user to create constraints by selecting values from options for each argument.

The main difference between the original conception, with the use of ADL, and the ARC Logic implementation is that the ARC Logic implementation encapsulates the translation from natural input to Prolog rules within the knowledge representation and inference system. Because all domain information is entered through the input methods, the format of the information is

predictable, so it can be checked, compared, and modified easily. The ability to compare constraints in this implementation is only possible because they are created in a uniform manner.

The conversion from input methods to Prolog rules is tightly intertwined with the knowledge representation and inference engine of the ARC Logic system, but this system can also potentially facilitate the use of a natural language processing component. The output of a natural language processing component to integrate with the ARC Logic system would be the information from the natural text in the form of ARC Logic input methods. A user using the ARC Logic implementation as a standalone program enters input methods to add domain information, as would a natural language processing (NLP) program or some other input component. All the tools of the ARC Logic system available to the user are available to the NLP program as well.

The defeasible reasoning system, with automatic rebutting and undercutting of constraints, pushes the knowledge representation much closer to the natural language description, allowing an easier or more accurate translation of the natural descriptions. A user can write an overgeneralizing constraint and later (or first) write and exception constraint without having to notice, much less deal with, the issue. When an NLP program extracts constraints from the natural text descriptions, it does not need to analyze if these constraints are overgeneralizing, because constraint comparison in the ARC Logic will take care of possible conflicts. Defeasibility is also a good way to handle "most," "some," "generally," and other terms that imply uncertainty. Certain indications in the text, or even headings of sections, can provide a reason for the NLP program to shift the focus of the description to certain sections or aspects of a Gothic cathedral, and the scope function facilitates this setting of context.

Finally, the ARC Logic system's approach to filling in missing information of a description can be beneficial for a natural language processing component. The shift away from the concept of the default model as a cathedral class and a specific cathedral as an instance of that class, to a more fluid inheriting hierarchy model, allows the NLP component to use multiple descriptions easily. The ARC Logic system allows multiple descriptions to be combined and checked for consistency in a manner that is no more complex than a single description. Combining different textual descriptions about specific cathedrals can create more complete logical models of those cathedrals. Conflicts arising in the combination of cathedral descriptions could highlight conflicts in the descriptions themselves, or highlight problems with the translation by the NLP program into input methods.

The use of descriptions, even with a natural language processing component, would likely need to coincide with some manually-created default model or models. A useful function of a natural language component would be the ability to take a large number of descriptions about general and specific Gothic cathedrals, and somehow combine their logical models to automatically create complete default descriptions. The ability to automatically create complete default descriptions for a domain would be very useful and have implications beyond the scope of the ARC project.

8.3 APPLICATION TO OTHER DOMAINS

The ARC Logic system is essentially domain-independent because, aside from some assumptions about the way objects and object-containment relationships work, all domain information comes from the user through input methods. The input methods limit the format in which information can enter the system, but there are no limitations on the information itself. A

user does not just have the ability to create and work with customized ontologies in the domain of Gothic cathedrals; the ARC Logic system can be used for any domain that is similar. Any domain which can be completely or shallowly expressed with the same types of domain information can be rendered by supplying the ARC Logic system the appropriate inputs. Anatomy, for example, is similar in many respects to architecture, and if one wanted to examine logical connections between the pieces of the body they could do so with the ARC Logic system.

One could also use the system to model things quite different from Gothic or bodily architecture, such as modeling sociopolitical interactions. Instead of architectural objects like columns, or bodily objects like bones, the ARC's objects could be used to designate different individuals, social groups, nations, cultures, occupations, economic classes, political parties, etc. Then the abundant relations between these groups/individuals/concepts could be created with constraints. In descriptions of social interactions between groups and individuals, the overgeneralization and specific exceptions are especially common. The defeasible nature of ARC Logic system would be a powerful way to represent the highly generalized, assumption-filled, domain of sociology and politics. The ability to use the input methods to write defeasible domain knowledge, particularly the creation of constraints in an inheritance hierarchy, could be very useful for logically modeling such a domain.

The ARC Logic system, whether it is used in the domain of Gothic cathedrals or some other, similar, domain, contains the functionality to create and use logical models of the domain to analyze natural descriptions. The ARC Logic system allows users, or other applications, to add domain-specific knowledge to create a custom ontology for a domain, as well as any number of default and specific descriptions, at different levels. These descriptions allow the user to model uncertain information, and conflicting rules and constraints are handled automatically by

the ARC Logic system. All of this information is added through a few input methods, which do not require expertise in Prolog or programming in general. The inference engine can be run, which adds all information that can be derived to the knowledge base, along with the certainty of each piece of information. This inference also checks for consistency, and ensures that only derivable information is still available in the knowledge base. This knowledge base of facts can then be queried or written to other output programs. The ARC Logic system is designed to be as expressive yet simple as possible; a goal it attempts by pushing the logical domain as close to natural description as possible. The ARC Logic implementation can be used directly in an intuitive manner, and forms a solid foundation of knowledge representation and inference for future extensions of the ARC project, as well as ventures into other related domains.

REFERENCES

[1] E.-E. Viollet-le-Duc, Dictionary of French Architecture from 11th to 16th Century, Paris: Libraries-Imprimeries R´eunies, 1856.

[2] C. Hollingsworth, S. Van Liefferinge, R. A. Smith, M. A. Covington and W. D. Potter, "Artificial Intelligence Techniques for Understanding Gothic Cathedrals," *Proceedings of the 2011 International Conference on Artificial Intelligence. Vol I. Worldcomp '11,* pp. 175-178, 2011.

[3] C. Hollingsworth, S. Van Liefferinge, R. A. Smith, M. A. Covington and W. D. Potter, "The ARC Project: Creating logical models of Gothic cathedrals using natural language processing," *Proceedings of the 5th ACL-HLT Workshop on Language Technology for Cultural Heritage, Social Sciences, and Humanities.,* pp. 63-68, 2011.

[4] W. J. Mitchell, The Logic of Architecture, Cambridge, MA: The MIT Press, 1990.

[5] L. Naish, Negation and Control in Prolog (Lecture Notes in Computer Science), Berlin: Springer-Verlag, 1986.

[6] M. A. Covington, D. Nute and A. Vellino, Prolog Programming in Depth, Upper Sadle River, New Jersey: Prentice Hall, 1997.

[7] M. A. Covington, D. Nute, N. Schmitz and D. Goodman, "From English to Prolog via Discourse Representation Theory," Advanced Computational Methods Center, The University of Georgia, 1988., 1988.

[8] G. Antoniou, "A Tutorial on Default Logics," *ACM Computing Surveys,* vol. 31, no. 3, pp. 337-359, 1999.

[9] J. L. Pollock, "Defeasible Reasoning," *Cognitive Science,* vol. 11, no. 4, pp. 481-518, October 1987.

[10] D. Nute, "Defeasible Prolog," *AAAI Fall Symposium on Automated Deduction in Nonstandard Logics*, pp. 105-112, 1993.

[11] D. Nute and M. Lewis, "A User's Manual For d-Prolog," University of Georgia, Athens, 1986.

[12] A. Cohen, A. J. Garc´ıa and G. R. Simari, "Backing and Undercutting in Defeasible Logic Programming," in *11th European conference on Symbolic and quantitative approaches to reasoning with uncertainty*, Belfast, 2011.

[13] BjornT, "Triforium Chartres," Internet: http://commons.wikimedia.org/wiki/File:Triforium_Chartres.jpg, 2006 [April 2012].

APPENDIX A

QUICK REFERENCE

INPUT METHODS:

TERM DEFINITIONS

define_relationship(+Name, +List)

List looks like [+Attribute1, +Attribute2, ...] or [ ] if no attributes.

Attributes are: transitive, reflexive, symmetric

Negative versions: intransitive, irreflexive, asymmetric

Defeasible versions: add a d_ to the beginning of any attribute

define_metarelationship(+Metarelation, +Relation1, +Relation2)

Metarelation can be: implies or antonym

define_object(subtype, +SubType, +SuperType)

define_object(supertype, +SuperType, +SubType)

CONSTRAINT CREATION

create_constraint(+Vx,+Cond,+Must,+Rel,+Min,+Max,+Type)

Overloaded versions of create_constraint:

create_constraint(Type1,Must,Rel,Min,Max,Type2)

Type1 is changed to two arguements: X, object(Type1, X)

create_constraint(Vx,Condition,Must,has,Type)

Min and Max both default to 1

create_constraint(Vx,Condition,Must,Rel,Type)

Min and Max both default to all

create_constraint(Type1,Must,Rel,Type2)

Uses Min and Max defaults in respect to Rel, and converts Type1

FACT ASSERTION

assert_fact(+X, +Defeasibility, -Asserted)

assert_fact(+X,+Defeasibility)

assert_fact(+X)

Defeasibility defaults to indefeasible

assert_object_instance(+Type, +IsPartOf, ?Name, +Defeasibility, -Asserted)

assert_object_instance(+Type, +IsPartOf, ?Name, +Defeasibility)

assert_object_instance(+Type, +IsPartOf, ?Name)

assert_object_instance(+Type, +IsPartOf)

MASS INFORMATION RETRIEVAL

all_relations(-Relations)

all_constraints(-Constraints)

all_facts(+Type, ?Def, ?Process, -Facts)

all_facts(?Def, ?Process, -Facts)

all_facts(-Facts)

all_facts_rm/4, all_facts_rm/3, all_facts_rm/1

Same as all_facts but does not include metadata.

APPENDIX B

TUTORIAL EXAMPLES

This appendix lists the input methods for each tutorial example used. Each line is entered in the Prolog query prompt. Alternatively, one or more .pl files can be saved and consulted that contain the following lines.

SIMPLE COLUMN EXAMPLE (From Chapter 3)

```
?- define_relationship(above, [transitive]).
?- create_constraint(column , must, has, capital).
?- create_constraint(column, must, has, shaft).
?- create_constraint(column, must, has, base).
?- create_constraint(capital, must, above, shaft).
?- create_constraint(shaft, must, above, base).
?- assert_object_instance(column, 0, ex_column).
```

EXTENDED COLUMN EXAMPLE (Chapter 4)

```
?- define_relationship(above, [transitive]).
?- define_relationship(immediately_above, [intransitive]).
?- define_metarelationship(antonym, above, below).
?- define_metarelationship(implies, immediately_above, above).
?- define_object(subtype, column, support).

?- create_constraint(support, must, has, shaft).
?- create_constraint(support, must, has, base).
?- create_constraint(column , must, has, capital).
```

?- create_constraint(column , d_must, has, necking).

?- create_constraint(capital, d_must, immediately_above, shaft).

?- create_constraint(capital, must, above, shaft).

?- create_constraint(capital, must, immediately_above, necking).

?- create_constraint(necking, must, immediately_above, shaft).

?- create_constraint(shaft, must, immediately_above, base).

?- assert_object_instance(column, 0, ex_column).

Possible additions and their results:

?- create_constraint(column , must, has, 1, 1, necking).

Creates indefeasible neckings for each column.

?- create_constraint(column , must, has, 0, 0, necking).

This can be defeasible or indefeasible. Removes the constraint for necking, so no necking is created for any column.

?- create_constraint(X, object(column, ex_column), must, has, 1, 1, necking).

This can be defeasible or indefeasible and any number of necking. Constraint only applies to this particular column.

?- assert_object_instance(necking, ex_column).

Asserts an indefeasible necking object that overrides the defeasible fact for the object.

CHARTRES BAY EXAMPLE

Moving up slightly in scope, the user can create a default bay in a nave. The term bay is used to designate a vertical slice, generally between columns, windows, or some other separating object. For this description a bay will be considered the space between the large supports that stretch from the ground to the very top of the interior of the nave. Figure 4 shows a bay in Chartres Cathedral.

Figure 4: Bay in Chartres Cathedral. **[13]**

This is an example description of the visible interior face of one bay in the nave of Chartres Cathedral.

```
?- set_scope(chartres).
?- create_constraint(bay, must, has, main_arcade_level).
?- create_constraint(bay, must, has, triforium_level).
```

```
?- create_constraint(bay, must, has, clerestory_level).
?- create_constraint(triforiumn_level, must, immediately_above,
main_arcade_level).
?- create_constraint(clerestory_level, must, immediately_above, triforiumn_level).

?- create_constraint(main_arcade_level, must, has, arch).
?- create_constraint(main_arcade_level, must, has, 2, 2, column).

?- create_constraint(triforium_level, must, has, 5, 5, column).
?- create_constraint(triforium_level, must, has, 4, 4, arch).

?- create_constraint(arch, must, above, 2, 2, column).

?- create_constraint(clerestory_level, must, has, 2, 2, lancet_window).
?- create_constraint(clerestory_level, must, has, rose).
?- create_constraint(clerestory_level, must, has, formeret_arch).
?- create_constraint(rose, must, above, lancet_window).
?- create_constraint(formeret_arch, must, above, rose).

?- assert_object_instance(bay, chartres).
```

This is a small example, and is of course one of an almost unlimited number of ways to describe a bay. Here the user could create 8 bays in the nave, and each bay object will contain the information for all 3 levels. Instead, the user could create 3 levels in the nave, and then describe the horizontal information in each level, not even using the concept of bays, creating a completely different hierarchy of the description.