THEORIES AND TOOLS FOR DESIGNING APPLICATION-SPECIFIC KNOWLEDGE BASE DATA MODELS

by
Mark Graves

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1993

Doctoral Committee:

Professor William Rounds, Chair
Associate Professor Michael Boehnke
Assistant Professor Edmund Durfee
Associate Professor John Laird
Assistant Professor Elke Rundensteiner

To Mom,

without whom I never

would have made it.

Acknowledgements

With any dissertation there are many people who played some part, and who were supportive in some manner. I would like to take this opportunity to thank those who contributed directly to the ideas and views presented here. First, I would like to thank my chair Bill Rounds for his guidance, encouragement and support and for teaching me to look at problems from different perspectives. I would like to thank all of those on my proposal and dissertation committee for their suggestions and comments and for introducing me to new areas of research: Mike Boehnke, Ed Durfee, John Laird, Steve Lytinen, Todd Knoblock, and Elke Rundensteiner. I would also like to thank several supportive students at the University of Michigan who commented on this work: Clare Bates Congdon, Peter Hastings, Stacie Hibino, Scott Huffman, Jeff Kirtner, Karen Lipinsky, Karen Mohlke, and Mark Young.

Some of the ideas and the interest in natural language processing grew while I was working for Rich Cullingford at Georgia Tech and Intelligent Business Systems. Leo Obrst and Brian Phillips at IBS also broadened my background in natural language processing and introduced me to some of the research upon which part of this dissertation is based. There were several students at Georgia Tech who helped me as I began the basis for this work and/or made suggestions as it started to take shape: Linda Gatti, Tom Hinrichs, Patsy Holmes, Joel Martin, Mike Redmond, Hong Shinn, Elise Turner, Roy Turner, and David Wood.

I would also like to thank the friends and colleagues who supported me in many ways as I struggled to finish this work. Thank you.

A portion of this dissertation was supported by NSF grant ISI-9120851.

Table of Contents

List of Figures

Figure

Chapter 1
Introduction

We use what we are and have, to know; and what we know, to be and have still more.
— Maurice Blondel

Until now, if someone wanted to design a new knowledge base, they had no alternative but to start from scratch. Currently, when developing an application which requires a knowledge base, most people use an existing knowledge base and coerce their entire application to fit the knowledge base — not because this is the best approach, but because this is the only approach. There is a strong need by those who develop knowledge-intensive applications for a flexible, extensible knowledge base which can represent knowledge in a manner which is natural to the application domain. But, this need has largely been ignored by the knowledge base community — ignored not because of disinterest but just because there were no theories powerful enough to solve the problem.

The most important design decision in developing a new knowledge base is choosing the best data model. A knowledge base stores complex information, and the data model must be capable of expressing the required knowledge. There are a variety of data models already available from databases, e.g. relational, hierarchical, semantic, object-oriented, and complex object. If developing a general-purpose knowledge base, which is to be used for a variety of tasks, this is a very difficult decision and is the focus of much knowledge base research. But, if the goal is a knowledge base which can be used effectively for a specific application, the solution is much simpler — use the structures that the designer already uses to manually solve problems in the application domain — the *natural* choice. If the designer does not already have a fixed, coherent, consistent, and appropriate collection of structures and methods for solving the problem, then the knowledge base data model must also be *flexible* and *extensible*.

To guide development of the application-specific knowledge base and its data model, we take advantage of sophisticated theoretical tools which have been proven effective in other areas of computer science and extend them to form a foundation for knowledge base design. We import formalisms from knowledge representation, natural language semantics, programming language research, constructive type theory, and databases. These form a strong, theoretical foundation for knowledge base design upon which we have implemented a knowledge base design tool called WEAVE.

1.1 Motivation

We envision WEAVE as the first step in developing a complete knowledge base development environment, where a prototype knowledge base can be developed for a new application quickly by specifying its internal (physical) representation and its data model, including type constructors and access methods. The prototype can then be refined as more reasoners, problem solvers, and querying methods are added to it or as either the developer's understanding of the domain changes or the domain itself evolves. Other projects such as CYC [LG90] or MKS [PT91] have similar goals of multi-faceted knowledge bases, but we have emphasized rapid prototyping of knowledge bases for specific applications rather than building huge architectures for common knowledge or enterprise integration. We assume the knowledge-intensive applications are implemented in traditional programming languages, and they could be a general problem solver, a natural language or machine learning system, a decision-support system, a knowledge base or database application program, an expert system, a scientific or manufacturing system, or any other knowledge-intensive application.

In our approach, the knowledge base design process is reduced to having the application developer give a high-level specification of an interface between the knowledge-intensive application and a knowledge base we provide; the knowledge base design tool WEAVE then creates the interface. This requires that the underlying knowledge base be expressive enough to represent the knowledge from the domain of interest, and that the high-level specification language be flexible enough to specify application-specific data models for a variety of tasks. Because the applications have complex requirements, we also require that the interface allow the knowledge base to define, manipulate, and retrieve knowledge using different views (not just retrieve as is meant by a database view).

By allowing different application interfaces to access the same knowledge base, knowledge sharing is enabled where possible. Although we allow for multiple applications each

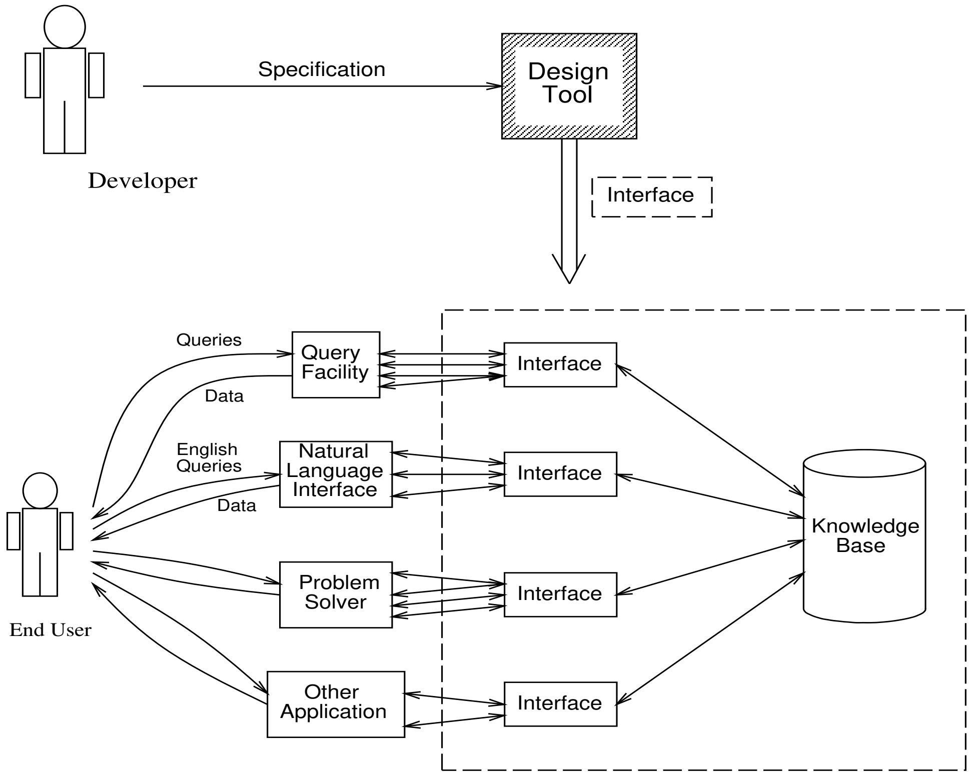


Figure 1: Designing Application-Specific Knowledge Base Interfaces

with multiple views on the same knowledge base, we restrict our study to nonconcurrent access by few end users. This is shown in Figure 1.

To make the knowledge base design process easier for the application developer, our design tool WEAVE provides a knowledge base, a high-level language for specifying how the knowledge is to be stored, and a language for specifying how the knowledge should be accessed by the application in terms of a data model. From the specification, WEAVE automatically creates the knowledge base interface to the application. Because WEAVE provides a persistent knowledge store (knowledge base), it is important that its representation be expressive enough to represent the required knowledge and flexible enough that the application developer can access the knowledge in various forms, each of which are a natural manner of representing the view of knowledge needed for the specific task. To meet the requirements of an expressive, flexible, natural representation which can be stored efficiently, we have chosen graphs. We have formalized higher-order, cyclic, directed graphs as a *graph logic* and implemented graph logic as a logic programming language, which we call WEB.

However as graphs become large, they become unwieldy and difficult to use. The solution is to break up the graphs into smaller pieces called *graph constructors.* The developer defines graph constructors as the representation and implements them as declarative logic programs in WEB.

However in a realistic setting, it is important to insure that the graph constructors are combined only in meaningful ways. This requires that the parameters and result of the graph constructor be typed. It is important that the type system be expressive enough for knowledge base design, extensible so new types can be added as the knowledge base is developed, and be compatible with the type system of the traditional programming language in which the application is implemented. To meet these requirements, we have modified constructive type theory [ML82] for knowledge base design. The application developer associates graph constructors in WEB with *data constructors* from constructive type theory. Data constructors create elements of an abstract data type, and abstract data types are created by type constructors. For example, LIST(A), where A is a type variable, is a type constructor which creates lists; LIST(SYMBOL) is an abstract data type for list of symbols which are created by the data constructors *pair* and *nil.* We have implemented a strongly-typed, functional programming language that incorporates constructive type theory, which we call SPIDER.

The application developer uses SPIDER to define type constructors in constructive type theory which create the abstract data types necessary for the application, and implements access methods in SPIDER on the data types. The developer collects the types and methods to form a knowledge base data model for the application. This data model is the specification for the application side of the knowledge base interface, and the graph constructors are the specification for the knowledge base side. WEAVE uses the specification to provide a mechanism for accessing the graphical knowledge from an application implemented in a traditional programming language, as specified by the knowledge base data model.

WEAVE sets up a translation from a data model representation, which can be manipulated by a traditional programming language, to a graphical representation of knowledge. Because we want the application to manipulate the knowledge in a natural way for each task, we require that this graphical representation can be translated to and from many different types (views) to be accessed by the application. This allows the application to use the knowledge in a manner which makes doing the task simpler and/or more efficient. This also allows different applications to share knowledge by using the same or overlapping graph constructors. The purpose of WEAVE is to provide a translation from a graphical representation of knowledge to a traditional programming language representation as specified by a data model which is one-to-many and reversible.

1.2 Contributions

Our methodology for knowledge base design provides a process for defining data models, which specify an interface to a knowledge-intensive application, and a general knowledge base for storing the knowledge. This is supported by formal theories that describe what can be done by the built-in knowledge base and by an implementation that creates prototype knowledge bases which have the user-defined data model.

The knowledge base design process is:

1. **Create a graphical sketch.** This should capture the structure and semantics of the knowledge for the application.
2. **Abstract common features of the sketch.** These abstractions (graph constructors) are sections of the graph that can be used to build and manipulate the graph in a meaningful way. They are specified in the declarative graph description language WEB.
3. **Group the abstractions into data types.** These graph abstractions (graph constructors) become data constructors for the type constructors which create application-specific abstract data types.
4. **Implement methods on the abstract data types.** These are implemented in the strongly-typed, functional programming language SPIDER.
5. **Collect the types and methods to form a data model.** These type constructors, abstract data types and access methods form the data model for the application's knowledge base.

We have applied this process to designing knowledge bases for use in problem solving, natural language processing, and molecular biology.

We propose an architecture with four layers for the knowledge base design tool, and theories at each layer to guide development. If after the knowledge base design process has stabilized, and there is a need for greater efficiency, then the lower levels of WEAVE can be replaced with a more efficient implementation which still has the functional and interface specification of the original, theory-guided design. It also appears that in many cases

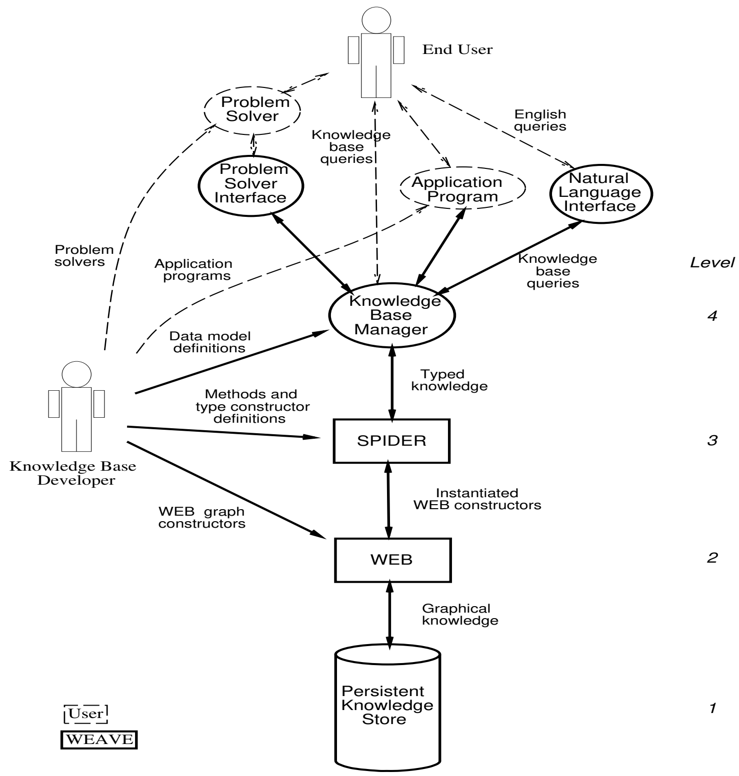


Figure 2: WEAVE System Architecture

more efficient implementations may be automatically compiled from the original theoretical definitions used to specify the application-specific knowledge base.

The architecture we have implemented consists of four levels: the *physical* (lowest) level, the *structural* level, the *data type* level, and the *data model* level. The physical level uses a binary logic, vivid knowledge store to organize the data and its abstractions. The structural level uses a *description theory* to define the structure of the knowledge in a persistent, structural (graphical) description language, called WEB. The data type level uses constructive type theory [ML82] to define the data types for the application in an extensible knowledge base programming language, called SPIDER. The fourth level uses an algebraic approach to define the data models. All four levels are combined into the implemented knowledge base design tool WEAVE as outlined in Figure 2.

In developing WEAVE, we have tried to:

- minimize the time necessary to design knowledge base data models;
- ignore run-time efficiency;
- make the theory and implementation conform to each other; and
- make the underlying representation of WEB very expressive.

To do this, the three hardest problems to solve were:

1. Finding a way of specifying views to define, manipulate, and access data with a complex structure. This was solved by using different graph constructors break up the graph in many different ways and retrieving data with a *graph querying* algorithm, which retrieves graphs from a knowledge base which match a partial specification in graph logic. This required formalizing graphs as a higher-order logic restricted to binary predicates, which forms the basis of WEB.
2. Constructive type theory was developed for mathematical proofs. It needed to be extended to talk about graph structures and be made easier to use. This was solved by adding new, built-in type constructors and developing inference rule construction algorithms which create the natural-deduction style inference rules which specify the user-defined type constructors.
3. Implementing SPIDER based on constructive type theory. This was solved by implementing inference rules by giving them an operational semantics.

To simplify our task we made two assumptions:

1. The natural representation of domain knowledge contains only symbolic and/or graphical data. Thus, WEB needs only to store symbolic and graphical data.
2. The application is implemented in a functional programming language, such as Lisp or SML [MTHM90].

This still allows for a wide variety of applications to be developed, and we extend the resulting theory and implementation at the points where it seems the most restrictive.

The symbolic, graphical representation consists of a description theory built upon a graphical foundation which can also be formalized as a higher-order logic restricted to binary predicates. This is implemented as the binary logic programming language called WEB.

The programming language which SPIDER, which accesses the representation, is a strongly-typed, functional programming language. Rather than develop a full programming language, we have developed a restricted functional language which contains a minimal set of functional constructs and can be embedded in the complete functional language, in which the application is implemented. Because most programming language paradigms, e.g. object-oriented or procedural, have a functional component, this approach will be applicable for most programming languages. The programming language we have implemented, SPIDER, uses constructive type theory as the foundation for its types.

The translation between the graphical representation of WEB and the programming language SPIDER takes place through type constructor definitions in SPIDER. But, rather than incur the cost of translating between the graphical representations in WEB and a different form in SPIDER, we have implemented SPIDER in such a manner that when SPIDER programs are executed, they manipulate the WEB graphs directly without any translation taking place. This is transparent to the user of SPIDER as it appears to work as any other functional programming language. Thus, the natural, graphical representation of WEB is both how the knowledge base developer thinks of the structural information and the foundation for the application's representation. This also makes it easier to develop the applications, because both the application and the end user are isolated from the details of the graphs, except for what is needed for the current task.

To access the graphs in WEB as SPIDER data constructors, there needs to be a mechanism for retrieving all graphs from the knowledge base which match a given partial specification. To access WEB's graphs through SPIDER's type system, constructive type theory must be able to reason with data types which have a theoretical analogue to WEB's graph logic. For the execution of SPIDER programs to be driven by the type system, we must give an operational semantics to the data types defined using constructive type theory.

These three requirements:

1. a mechanism for retrieving graphs from a persistent knowledge store which match a partial specification,
2. extensions to constructive type theory and the creation of new type constructors which allow data types to be created that have a structure analogous to graphs, and
3. an operational semantics for data types created by constructive type theory

are the primary technical contributions developed in this dissertation. These contributions, along with the novel integration of theoretical and practical techniques from knowledge representation, natural language semantics, programming languages, and databases, are used to implement the knowledge base design tool WEAVE.

We formalize WEB as a graph logic by building a description theory which presents the constructs in WEB in both graphical and logical terms. This allows us to define a new algorithm called *graph querying* which retrieves all graphs from a knowledge base which match a partial specification as expressed in graph logic.

Constructive (intuitionistic) mathematics is a non-classical approach which does not allow for indirect proofs. Constructive type theory [ML82] encodes logical propositions as types in a formalism which allows mathematical proofs to be tightly coupled to computer programs. It uses natural deduction style inference rules to develop and reason with types in a manner which is both mathematically rigorous and computationally perspicuous.

Most uses of constructive type theory have been in automated reasoning systems [CAB+86, Pau89] where a general theorem prover is used to prove theorems in constructive type theory, usually with human guidance. Because the proof is constructive, it is possible to extract a program from the proof. We ignore the theorem proving aspects of constructive type theory and use the theory directly in the execution of proofs. A type constructor is defined in constructive type theory by a collection of natural deduction style inference rules. Instead of using these rules in an automated reasoner, they are used in SPIDER as a computational engine for the evaluation of functional programs.

The advantages of constructive type theory are:

- A type discipline organizes the data and can let us manipulate it more efficiently.
- It is powerful enough to express the types necessary for knowledge base design.
- Types can be organized in a manner which allows for inheritance.
- A constructive type system lets properties of the type definitions be implemented.
- Algorithms can be developed which automatically construct inference rules within the theory. We have done this for type constructors which are useful for knowledge base programming.
- As implemented in SPIDER, it abstracts the representation, in WEB, and isolates the end user from the structural details. This separates the type information from the structure information and leads to a cleaner notion of inheritance.
- The operational semantics we have developed generates proofs of correctness which yield an extra layer of certitude that a program meets its specification.
- Its inference rules can be used to define methods similar to what is available for object-oriented programming.

Now we will consider as an example, the data type BinaryTree which consists of nodes and leaves where all data are contained in the leaves. We can prove many properties on the type:

- All binary trees are finite. (Because they are constructed by a finite application of *node* and *leaf*.)

- The expression $node(leaf(a), leaf(b))$ is an element of BinaryTree.
- The top-level construct in a binary tree is either a *leaf* or a *node*.
- The tree searching function returns *true* when given $node(leaf(a), leaf(b))$ and a as arguments — the function is defined by the lambda expression:

$$\lambda x.\lambda ele.BinTree\text{-}elim(x, \lambda a.(ele \text{ EQUAL } a), \lambda l.\lambda r.\lambda hl.\lambda hr.(hl \text{ OR } hr))$$

Although they are all interesting properties, we will emphasize properties like the last one in this work.

Because constructive type theory has been used primarily as a basis for mathematical proofs, it is necessary to modify it for it to be applicable to knowledge base design. For example, the graphs in WEB are allowed to have *multi-valued attributes*, where multiple arcs with the same label can originate at one node. When a data constructor is defined using a multi-valued attribute, one instance of the data constructor can refer to multiple, simultaneous occurrences of graphs in the knowledge base. This can be used to define set-like types. Because types can be defined using graph constructors with a much more complex structure than a multi-valued attribute, we have developed a generalized notion of set-valued data constructors called *inductive types*. We modify constructive type theory to handle inductive types by introducing set-valued variables to the inference rules which range over subsets of a type and by introducing induction variables which work analogous to recurse variables in recursive types to refer to the computation which remains in obtaining the desired, canonical form. We also have developed a type constructor which creates a modified cartesian product of two types which can be used to create binary functions in a manner analogous to unary ones. This allows methods over multiple types to still be associated with one (product) type which lends itself to a much stronger organization of types and methods. It also can help in specifying data model definitions.

To make constructive type theory useful, we have developed algorithms which automatically create all the inference rules needed for a type constructor when given a type definition in SPIDER. This is possible because of the restrictions that are placed on the type constructors which can be formed. Although these restrictions allow for a wide variety of knowledge base types to be defined, they still are very restrictive in terms of the theoretical expressiveness of constructive type theory. We have developed algorithms for the allowed type constructors in SPIDER: simple, recursive, inductive, product, and all combinations of them.

These modifications to constructive type theory, the algorithms which automatically construct inference rules, and the formalization of an operational semantics for the inference rules allow for the flexible and powerful definition of types for knowledge base design. When combined with the structural definition of graph logic and our graph querying algorithm, this leads to a system for specifying both structural and type information. This meets our goal of accessing a natural, graphical representation of knowledge through a traditional, functional programming language, which allows for the design of application-specific knowledge bases.

1.3 A More Substantial Application

We have developed knowledge bases within several areas in computer science including general knowledge representation, problem solving, and natural language processing, with positive results. However, we wanted a realistic, complex problem on which to demonstrate our work, and we have found the problem of mapping the human genome to be greatly in need of direct knowledge base support. Currently, there are many different approaches to build genome maps at different levels of granularity, with different properties, and with different ways in which they are useful. Each map is based on laboratory procedures which can have errors and inconsistencies. Different statistical methods are used to deal with the problems, and they are based on different assumptions and models. People can generally deal with one kind of map at a time, though it is tedious. When multiple, heterogeneous maps are available, it can be difficult to handle the complexity.

We show that our general process for designing knowledge bases can be used for building a data model including multiple types of genome maps. We demonstrate the process on a simple representation for distance information in the genome and explain how queries can be asked of the knowledge base. We also show how order information can be represented in a similar fashion. Even at this preliminary stage, the results have proven to be useful and extremely promising for solving difficult problems in molecular biology.

Integrating heterogeneous maps is an especially good problem on which to demonstrate this approach because there is already an underlying structure (the genome) which people view in different ways (physical and genetic maps). This is not to say that the most computationally efficient way of representing the underlying structure of the maps will correspond to the genome, but merely indicates that there is a common structure to the maps, and this can guide development toward a more effective implementation. It gives us a place to start and fixes the user's view to be the heterogeneous maps. This results in the goal to find a common structure which can be efficiently used to integrate the information contained in multiple, heterogeneous maps.

One advantage of a knowledge base over an *ad hoc* system is the ability to query against it. Because we want the knowledge base to be useful in a realistic setting, it is also important to make the interface as user-friendly as possible. Query processing is done in WEAVE through a simple knowledge base manager. Currently, the knowledge base manager is given a partially instantiated data constructor and retrieves the structures in the knowledge base which match it.

Although this is work in progress, we want to set the context in which knowledge base design is most useful. We are developing a natural language interface to the knowledge base manager which will allow for English queries to the knowledge base such as:

Find the distance between marker D21S1 and marker D21S11.

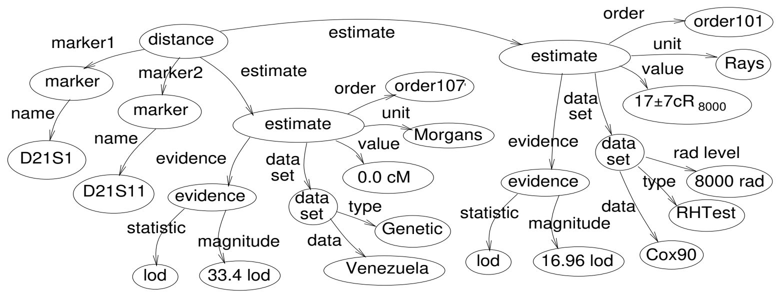
Find the best orderings.

Find order evidence for markers D21S16 and D21S48.

WEAVE is being used to implement the natural language interface, and this natural language interface application will also serve as another test and demonstration of WEAVE's effectiveness.

WEAVE can answer these queries and others like them now when expressed as data constructors queries in the knowledge base manager. The natural language queries and data constructor queries have a similar form which can be used in a unification-based natural language interface [Shi86]. The disadvantage in all implemented systems except ours is that this restricts the queries to have a form similar to the data constructors that were used to define the knowledge base. In WEAVE it is possible to have multiple, overlapping type definitions on the same structure. This allows data to be entered using one view of the structure and retrieved using alternative views. We show how data models can be created for DISTANCE and ORDER in chapter 5.

Distance between markers in a map can be represented graphically as a *distance* node with estimates of the distance represented as values of a multi-valued attribute (set-valued role) labeled *estimate*. In our graphical representation, a multi-valued attribute is denoted by multiple arcs with the same label originating at the same node. These estimates should be thought of as being collected by the units of the distance estimates. For example, the distance between the markers D21S1 and D21S11 from a genetic linkage map [THW+88] and a radiation hybrid map [CBP+90] may be represented as:



Abstractions, data constructors, and data types can be generated from this sketch as follows. First, find the sections of the graph which are likely to be reused in a semantically meaningful manner. In this example, the concepts involved are: distance, marker, estimate, evidence, and data set. Each of these concepts are associated with a section of the graph. We then define a graph constructor to build each section. When we separate the sections of the graph, we are left with five graph constructors which build the graphs. Each of these graph constructors is associated with a data constructor for a user-defined type. The data constructor's parameters are typed and accessed through SPIDER, and the graph is created when the data constructor is evaluated within SPIDER. Thus, these data constructors can be used to build a knowledge base. The knowledge base is accessed by functions which are defined on the data types, and the function's execution is specified by a collection of inference rules in constructive type theory.

These graph abstractions and data constructors allow the knowledge base to be built and queried against in a much more organized fashion than any existing semantic network or terminological subsumption architecture.

There are several advantages to designing a knowledge base to represent heterogeneous mapping information, which we discuss in more detail in chapter 5. The formalisms we describe here have proven themselves expressive enough for a wide variety of tasks and appear sufficiently powerful to help solve the problem of integrating heterogeneous maps, and because these formalisms are very flexible yet can be implemented efficiently, they promise to be a useful tool for mapping the human genome.

1.4 Plan of Thesis

Because our work is geared toward application-specific knowledge bases, it is important to both describe our results and demonstrate it on specific applications. Before explaining the applications of our work, we give an overview of our results in knowledge base design and give the technical contributions on the theory behind WEB and SPIDER. We demonstrate our techniques on a realistic problem in molecular biology, develop a simple problem solver to solve logic puzzles which require a domain specific representation, and show how a natural language interface can be developed to access our knowledge base. We also show representation schemes which we have developed using WEAVE which are useful for general knowledge representation, natural language semantics, and object-oriented databases. We then discuss related work in programming languages, databases, knowledge representation, and natural language semantics.

Chapter 2 describes briefly the key parts of the three higher levels in WEAVE's architecture. WEB is presented as a semantic knowledge base, and we describe it as a persistent graph logic programming language. The key technical contribution of WEB is a *graph querying* algorithm which uses graph unification to retrieve data from the knowledge base which matches a given specification. Constructive type theory is the theoretical foundation for SPIDER and it is explained in section 2.2. We then explain our algebraic approach to data models and give two example data models using it. One is $\mathcal{ALRC}$ which contains the key aspects of KL-ONE [BS85] and demonstrates that terminological subsumption languages can be described using our approach. The other example is a data model for situation theory [BE90a] which shows the flexibility and expressiveness of WEAVE.

Chapter 3 contains the details of WEB. It gives the graph querying algorithm with examples to explain its use. We also formalize WEB in terms of a graph logic. We give a definition for labeled graphs in terms of vertices, edges, and labels and show how the graphs are built using the logic incorporated in WEB. In section 3.3, we give an overview of the persistent knowledge store which forms the fourth (lowest) level in WEAVE's architecture.

Chapter 4 has the details of SPIDER. It shows how type inference rules can be used as a programming language, explains the type constructors which can be defined in SPIDER and gives algorithms to calculate their inference rules. We give an operational semantics for SPIDER in terms of constructive type theory inference rules and introduce some of the advantages of using constructive type theory to describe inheritance.

Chapter 5 describes the application of our knowledge base design process to a real problem in human genetics. We give the results we have obtained for integrating distance and order information from heterogeneous genome maps.

Chapter 6 contains application of our knowledge base design process to developing representation schemes for complex objects and feature structures from object-oriented databases and natural language semantics, respectively. We describe a simple constraint-based problem solver we have implemented and show how it uses an application-specific knowledge base to solve a logic puzzle. We then show how a natural language interface can be built on a knowledge base developed using WEAVE.

Chapter 7 gives related work in programming languages, databases, knowledge representation, and natural language semantics. We also describe some of our contributions to these areas.

Chapter 8 is the conclusion. We summarize our contributions and discuss possible extensions to this work.

Chapters 3 and 4 both depend upon understanding the information in chapter 2. Chapter 5 is independent of other chapters and contains all the background material necessary for understanding it. Most of the sections in chapters 6 and 7 are fairly self-contained, and any previous chapter would serve as sufficient background for them. The exceptions are the sections in chapter 6 on complex objects and feature structures which depend upon a familiarity with the inductive types of chapter 4.

Chapter 2
Definitions and Descriptions

The self is a relation which relates itself to its own self, or it is that in the relation [which accounts for it] that the relation relates itself to its own self; the self is not the relation but [consists in the fact] that the relation relates itself to its own self.
— *Søren Kierkegaard*

WEAVE is useful both for designing knowledge bases and developing prototypes of them. An application-specific data model can be specified in WEAVE without a great deal of unnecessary effort. It can then be changed as the designer's understanding of the application evolves. WEAVE simplifies the task by organizing the knowledge and giving access through knowledge base queries and methods. It is always possible to implement a knowledge base from scratch: it is just easier to not.

One of the strengths of our approach to knowledge base design is the separation of type information from structure information in the knowledge base. This allows each to be developed in accordance with its own constraints with a minimum of unnecessary overlap. WEB and SPIDER are each useful contributions, but when combined, this novel approach yields a dramatic improvement in the possible knowledge base designs. This occurs because each type can be represented as different structures and each structure can be abstracted as different types. The increased combination of type/structure interactions allow for more flexible design and the natural sharing of common type or structure information where appropriate. This eliminates redundancies and possible inconsistencies in the knowledge base and can allow multiple problem solvers to be used because they share the data stored in the structure, but they can access it in the manner best suited to that kind of problem solving.

WEAVE's implemented architecture consists of four levels: the *physical* (lowest) level, the *structural* level, the *data type* level, and the *data model* level. The physical level uses a binary logic, vivid knowledge store [EBBK89, DK79] to organize the data and its abstractions. The structural level uses a *description theory* to define the structure of the knowledge in a persistent, structural (graphical) description language called WEB. The data type level uses constructive type theory [ML82] to define the data types for the application in an extensible knowledge base programming language called SPIDER. The fourth level uses an algebraic approach to define the data models. All four levels are combined into the implemented knowledge base design tool WEAVE.

We describe briefly the key parts of the three higher levels in WEAVE's architecture. WEB is presented as a semantic knowledge base, and we describe it as a persistent graph logic programming language. The key technical contribution of WEB is a *graph querying* algorithm which uses graph unification to retrieve data from the knowledge base which matches a given specification. This is used both for knowledge base querying and as the interface between WEB and SPIDER; it is explained in section 2.1. Constructive type theory is the theoretical foundation for SPIDER, and it is explained in section 2.2. We then explain our algebraic approach to data models and give two example data models using it in section 2.3. One is $\mathcal{ALRC}$ which contains the key aspects of KL-ONE [BS85] and demonstrates that terminological subsumption languages can be described using our approach. The other example is a data model for situation theory [BE90a] which shows the flexibility and expressiveness of our approach. Situation theory is a theory of information content which supports general, heterogeneous inferencing. The persistent knowledge store (fourth level) is not discussed until section 3.3.

2.1 Semantic Knowledge Base

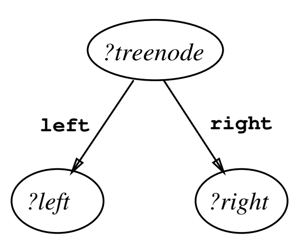
WEB has a graphical framework based upon semantic networks. It combines aspects of knowledge representation languages [MBJK90], feature structures [KR86, Car92], ψ-types [AK84] (which are a foundation for terminological subsumption languages [BS85]), semantic data models [HK87, PM88], and binary logic programming [DK79, BL87]. It also has aspects similar to Conceptual Graphs [Sow84], but organizes higher-order constructs differently. Most logic-based systems only consider first order predicate calculus as a logical foundation. WEB may be modeled as a higher-order predicate logic restricted to binary predicates. WEB uses graph querying for knowledge base access and does not do classification for terminological reasoning [BS85, BBMR89].

The emphasis on binary predicates is an old one which showed the relationship between semantic nets and predicate logic then was quickly dropped in favor of n-ary predicates because a logic based on binary predicates was unwieldy. However, there are two advantages in returning to binary logic for WEB. The first is that it forms a simple foundation which can be manipulated automatically. This is very important for extensibility. There is also not the original disadvantage of unwieldiness because the end user does not deal directly with binary logic but uses it only through SPIDER. The second advantage is that it is easy to treat the binary predicates as attributes in semantic nets, roles in frames, arcs in graphs, etc. This allows the designer of the types in SPIDER a natural foundation upon which to develop application-specific types.

Binary data models have been examined for semantic databases. One particularly similar data model to WEB is also one of the earliest: the *semantic binary data model* [Abr74] tried to have a minimal set of primitive constructs from which to build more powerful structures. This later led to the development of the NIAM (Nijssen Information Analysis Methodology) data model [VB82] which has influenced conceptual schemas in relational databases and led to the development of other binary data models [Mar83, Ris85, Ris86]. Binary formalisms have also been used in a graphical framework for other databases [PPT91, GPG90, CCM92]

2.1.1 Graph Logic Programming

It is sometimes useful to associate a function which creates new nodes in the knowledge base with a WEB program. We refer to these graph building programs as WEB constructors. The created nodes can then be bound to WEB variables and used as part of a parameterized sequence. For example, the WEB constructor `treenode` creates a new graph node which is bound to the variable *?treenode* and creates the arcs `left` and `right` coming from it, then returns *?treenode* as the result of the WEB program, where ?⟨*name*⟩ denotes a variable.

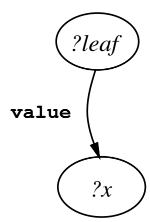


This can be used to build binary trees where $?left$ and $?right$ are bound to either leaves or treenodes. This is defined by

$$treenode(?left, ?right) \equiv [\text{create } ?treenode]\ (left\ ?treenode\ ?left)$$
$$(right\ ?treenode\ ?right)\ [\text{return } ?treenode]$$

The `treenode` WEB program is passed two arguments $?left$ and $?right$. It creates a new node in the graphical knowledge base and binds it to the variable $?treenode$. Then, arcs are created with labels `left` and `right` which point to the graph nodes bound to $?left$ and $?right$, respectively. The node which was created and bound to $?treenode$ is returned as the value of the WEB program.

Now, we can define the WEB constructor *leaf*, which creates a new node in the graph and connects it to the constructor's one argument via a new arc called *value*. The constructor then returns the new node in the graph.



This is defined by

$$leaf(?x) \equiv [\text{create } ?leaf]\ (value\ ?leaf\ ?x)\ [\text{return } ?leaf]$$

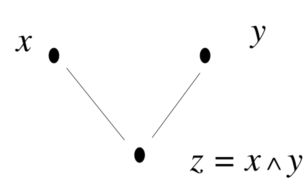
These two constructors can be used to build tree-like structures in the WEB knowledge base by associating them with the data constructors `node` and `leaf` in the SPIDER type BinaryTree.

2.1.2 Graph Querying

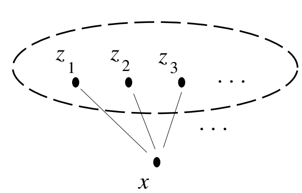
Graph querying is used to retrieve data from the knowledge base. This is useful both for general *ad hoc* queries and in developing access and simple reasoning methods in SPIDER. Because graphs in the WEB knowledge base usually contain more information than is associated with an individual constructor, graph querying is used to retrieve only the necessary information. In the example of the previous subsection, when a SPIDER method is executed on a binary tree, graph querying is used to obtain either the left and right or the leaf data value as appropriate.

WEB can be considered as an attributive description formalism [NS90]. Currently, there are two predominant attributive description formalisms: terminological subsumption languages, which are derived from KL-ONE [BS85], and feature structures, which evolved in computational linguistics [KR86, Car92, Shi86]. WEB uses a relational approach to defining multi-valued attributes (similar to the binary roles of terminological subsumption languages) but uses *graph querying* as the primary processing paradigm, rather than classification or graph unification as used in terminological subsumption languages or feature structures, respectively. Terminological subsumption languages are usually described as inference, but classification can also be defined in terms of feature graphs.

Graph unification, classification, and graph querying are all related. Consider a partial order on feature graphs $\langle \mathcal{F}, < \rangle$ which is defined by "graph subsumption".[1] The graph unification problem is: given $x, y \in \mathcal{F}$, find the *most general unifier* $z \in \mathcal{F}$ such that $z < x$ and $z < y$, and there are no (other) unifiers $z' \in \mathcal{F}$ such that $z < z'$. This is written $x \wedge y$.

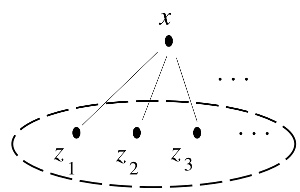


The classification problem is: to compute the subsumption hierarchy of a set of terminological definitions $\mathcal{T} \subset \mathcal{F}$. That is, given $x \in \mathcal{F}$, find the most *specific* $Z \subset \mathcal{T}$ such that $x < z_i,\ z_i \in Z$.



[1] We define the ordering with the more general (but less informative) concept as the greater one.

The graph querying problem is: given $x \in \mathcal{F}$ and a $KB \subset \mathcal{F}$, find the most *general* $Z \subset KB$ such that $z_i < x,\ z_i \in Z$.



For example, consider the concrete relations of *parent* and *gender*. We can define them as attributes in the knowledge base. Then, if we define five instances in the set KB:

$(parent\ \ Fred\ Tom)$	$(gender\ Tom\ \ \ male)$
$(parent\ \ Fred\ Mary)$	$(gender\ Mary\ female)$
$(gender\ Fred\ male)$	

we can query against the knowledge base [query $(\mathit{parent}\ \ \mathit{Fred}\ \ ?\mathit{child})$] which will return a binding of $?\mathit{child}$ to *Tom* and *Mary*. In this example, x is the feature graph $(\mathit{parent}\ \ \mathit{Fred}\ \ ?\mathit{child})$ amd the solution set Z has two elements $(\mathit{parent}\ \ \mathit{Fred}\ \ \mathit{Tom})$ and $(\mathit{parent}\ \ \mathit{Fred}\ \ \mathit{Mary})$.

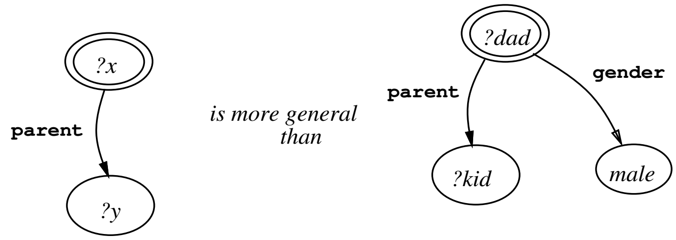
To compare classification and graph querying consider the definition of the WEB program `father`:

$$\mathit{father}(?\mathit{dad}, ?\mathit{kid})\ \equiv\ (\mathit{parent}\ ?\mathit{dad}\ ?\mathit{kid})\ (\mathit{gender}\ ?\mathit{dad}\ \ \mathit{male})\ [\text{return }?\mathit{dad}]$$

If the set of terminological definitions contains `parent-concept` where

$$\mathit{parent\_concept}(?x, ?y)\ \equiv\ (\mathit{parent}\ ?x\ ?y)\ [\text{return }?x],$$

then classifying the sequence for father shows that $\{\mathit{parent\_concept}\}$ is the most specific set of feature graphs which is more general than *father*, i.e.,

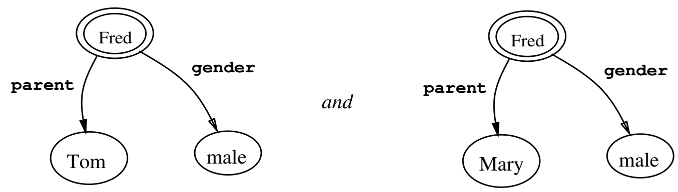


where the double circle denotes the value returned. If the definition

$$\mathit{man}(?x)\ \equiv\ (\mathit{gender}\ ?x\ \ \mathit{male})\ [\text{return }?x]$$

were in the set Z, classification would have included `man` in the resulting set, too.

When graph querying is given the feature structure for *father*, it finds the graphs



because these are the most general graphs in the knowledge base which are more specific than the query feature structure. The resulting answer is either the set of tuples $\{father(Fred, Tom), father(Fred, Mary)\}$ or the set of node(s) $\{Fred\}$, depending upon how the query was set up.

2.2 Constructive Type Theory

The SPIDER types are defined by type constructors in constructive type theory. A type constructor is specified by a collection of *data constructors* with corresponding graph primitives (in the knowledge base) which are manipulated as the data constructor is manipulated. Each data constructor may be manipulated only in accordance with its logical inference rules. This formalizes exactly how a type may be manipulated by giving it a firm, logical basis.

The type constructors can be instantiated into new abstract data types. For example, the type constructor CARTESIAN PRODUCT [ML82] can be combined with the type constructor SET (section 4.2.2.2) and instantiated with the abstract data type STRING as SET(STRING×STRING×STRING). A type constructor is defined by a collection of four kinds of inference rules. The four kinds of inference rules are: a formation rule, some introduction rules, an elimination rule, and some computation rules.[2]

A new type constructor is defined by writing a *formation inference rule* in constructive type theory. This tells how the type constructor is parameterized and how instances of it can be formed. For each data constructor in the type, an *introduction inference rule* is specified which tells how the elements of the instantiated type constructor (abstract data type) can be formed. For example, LIST(A) has data constructors *null* and $cons(a, l)$ where a is an element of the type A and l is recursively defined to be in *List(A)*. Each data

[2] In the theory there are also congruence rules, which are explained in section 4.4 where they are used. Congruence rules are not as prevalent in SPIDER as other systems based on constructive type theory, because of the flexibility in defining structure in WEB and the overlap of types. This eliminates most of the need for them. They are still required to set up inclusion polymorphism as described in section 4.4.

constructor is associated with a graph constructor which creates an appropriate entry in WEB.

In SPIDER, the user defines a formation rule and the introduction rules, and the system computes an elimination rule and appropriate computation rules from them [Bac86b] making use of some simplifying assumptions, e.g., the type constructors are of one of four kinds (see chapter 4). The elimination rule abstracts how to perform computations on the type, and the computation rules prescribe how to evaluate instantiations of the elimination rule, which are defined by functions (programs) on the type. Since an element of a type can be formed only through the data constructors, there is one computation rule for each introduction rule. (The algorithms for computing the elimination and computation rules are described in section 4.2.)

2.2.1 Type Inference Rules

The type inference rules tell how to reason with a type. For example, consider the inference rules for the familiar data type for binary trees. The BinTree(A) type is used to explain the structure of the rules, while similar steps would be used to define other types.

For simplicity, all data in the tree is kept in the leaves. To define the type BinTree(A), the user must define a formation rule and the introduction rules for the type. The formation rule defines the parameters of the type. There is only one parameter, which is the type variable A, which is instantiated to form abstract data types. There is an introduction rule for each data constructor in the type. The BinTree type has two inference rules corresponding to its two data constructors, one for `leaf` and one for `node`. From the formation and introduction rules, SPIDER computes an elimination rule and appropriate computation rules. Constructive type theory requires that these rules exist to reason on the type, and because of the restrictions SPIDER places on the types, it is possible to compute these rules automatically. This is what gives SPIDER a lot of its power. The elimination rule abstracts how to perform computations on the type, and the computation rules prescribe how to evaluate instantiations of the elimination rule, which are defined by functions (programs) on the type.

The natural deduction inference rules for binary trees are given below, suppressing all extraneous assumptions. We use $x \in T$ to denote that x is an element of the (constructive) type T.

Formation Rule: The *formation rule* tells how to form the type. If A is a type, then it can be inferred that *BinTree(A)* is a type, where A is a type variable [CW85].

BinTree-formation

$$\frac{A\ type}{BinTree(A)\ type}$$

The BinTree formation rule states: if A is a type, then *BinTree(A)* is a type.

Introduction Rules: An *introduction rule* defines how members of the type can be introduced. The data type BinTree is constructed through two data constructors: **leaf** and **node**. Each data constructor has an introduction rule to introduce its existence for the type.

leaf-introduction

$$\frac{a \in A}{\text{leaf}(a) \in BinTree(A)}$$

If a is a member of A, we can conclude that **leaf**(a) is a member of the type *BinTree(A)*.

node-introduction

$$\frac{l \in BinTree(A) \qquad r \in BinTree(A)}{\text{node}(l, r) \in BinTree(A)}$$

If l and r are members of the type *BinTree*(A), then **node**(l, r) is a member of the type *BinTree*(A).

Elimination Rule: Each type has an *elimination rule* which tells how to reason over members of that type. *BinTree-elim* is used to define functions over binary trees. Functions are defined by specifying what expression each data constructor should yield. The important part of the elimination rule (and computation rules) for this system is the conclusion of the rule(s). As type inference is not done in the system, the premises are only used to correctly construct the elimination function BinTree-elim.

The conclusion of the elimination rule is:

$$\textit{BinTree-elim}(x, leaf\_abs, node\_abs) \ \in \ C[x].$$

BinTree-elim is a form of three arguments, and it is in the type $C[x]$, specifically in the class of objects generated by the first argument. The second and third argument specify how to calculate a certain value when given x. How the arguments $leaf\_abs$ and $node\_abs$ are used is defined by the computation rules. The elimination form is evaluated using lazy, normal-order reduction.[3] The complete elimination inference rule is:

BinTree-elimination

$$\frac{\begin{array}{ll} [\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] & \text{— type premise} \\ x \in BinTree(A) & \text{— major premise} \\ [\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\text{leaf}(a)]]\!] & \text{— leaf premise} \\ [\![\ l \in BinTree(A) & \text{— node premise} \\ \quad r \in BinTree(A) & \\ \quad rec\_l \in C[l] & \\ \quad rec\_r \in C[r] & \\ \quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\text{node}(l, r)] & \\ ]\!] & \end{array}}{BinTree\text{-}elim(x, leaf\_abs, node\_abs) \ \in \ C[x]}$$

The elimination rule has four assumptions. Three of them are conditional. The first two, the *type premise* and the *major premise* are similar in all SPIDER elimination rules. The other two are dependent upon the introduction rule. The expression $C[w]$ refers to the class generated by the type (indexed by objects in the type).

The *type premise* defines the class generated by the type (indexed by objects in the type), and the *major premise* specifies an arbitrary element of the type to be reasoned with. The $leaf\_abs$ and $node\_abs$ terms will be defined by the individual programs on the type, but they must be of the type specified in their respective premises.

The *leaf premise*

$$[\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\text{leaf}(a)]]\!]$$

is a hypothetical rule (denoted by $[\![\ premises \quad \triangleright \quad conclusion \ ]\!]$) which states the form $leaf\_abs(a)$ is in the class generated by $leaf(a)$.

This notation for inference rules comes from Backhouse [Bac86a] and is similar to the notation used by Dijkstra [DF84]. We use it in this dissertation because it makes clearer the description of the algorithms which calculate the inference rules and the description of how constructive type theory is modified for knowledge base design.

---

[3] All computations in the system use lazy evaluation and normal-order reduction, unless it can be proven that the eager, applicative-order reduction yields the same result.

The *node premise*

$$
\begin{array}{l}
[\![\quad l \in BinTree(A) \\
\quad r \in BinTree(A) \\
\quad rec\_l \in C[l] \\
\quad rec\_r \in C[r] \\
\quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \;\in\; C[\mathrm{node}(l, r)] \\
]\!]
\end{array}
$$

is a hypothetical rule (denoted by $[\![\ premises \quad \triangleright \quad conclusion\ ]\!]$) which states the form

$$node\_abs(l, r, rec\_l, rec\_r)$$

is in the class generated by $node(l, r)$.

If there were a data constructor in the type which had no arguments, say *empty*, then the **BinTree-elimination** rule would have an additional premise, *empty-premise*, of the form $empty\_val \in C[\mathrm{empty}]$. This states that $empty\_val$ is in the class of results generated by the "empty" element. This occurs because the introduction rule for *empty* has no premises; it only has the conclusion empty $\in BinTree(A)$.

In a more traditional notation, suppressing all extraneous assumptions, the elimination rule looks like:

BinTree-elimination

$$
\dfrac{\dfrac{w \in BinTree(A)}{C[w]\ type} \qquad x \in BinTree(A) \qquad \dfrac{a \in A}{leaf\_abs(a) \;\in\; C[\mathrm{leaf}(a)]} \qquad \dfrac{\begin{array}{ll} l \in BinTree(A) & rec\_l \in C[l] \\ r \in BinTree(A) & rec\_r \in C[r] \end{array}}{node\_abs(l, r, rec\_l, rec\_r) \;\in\; C[\mathrm{node}(l, r)]}}{BinTree\text{-}elim(x, leaf\_abs, node\_abs) \in C[x]}
$$

or with additional assumptions, Γ, as:

$$
\dfrac{\begin{array}{l}
\Gamma,\ w \in BinTree(A)\ \vdash\ C[w]\ type \qquad \Gamma\ \vdash\ x \in BinTree(A) \\
\Gamma,\ a \in A\ \vdash\ leaf\_abs \in C[\mathrm{leaf}(a)] \\
\Gamma,\ l \in BinTree(A),\ r \in BinTree(A),\ rec\_l \in C[l],\ rec\_r \in C[r] \\
\qquad\qquad \vdash\ node\_abs(l, r, rec\_l, rec\_r) \in C[\mathrm{node}(l, r)]
\end{array}}{\Gamma\ \vdash\ BinTree\text{-}elim(x, leaf\_abs, node\_abs) \in C[x]}
$$

Variables ending with *_abs* are bound to lambda abstractions. The variable *leaf_abs* is bound to the lambda abstraction defined by a user program which tells what the (functional) program should return, if passed an element in BinTree(A) of $leaf(a)$. The abstraction *leaf_abs* has one argument which is bound to the parameter of *leaf*. The variable *node_abs* is bound to the lambda abstraction which specifies the result for nodes. The arguments to *node* are given as the first two parameters, l and r, in *node_abs*, but because the introduction rule specifies that *node* is recursive in both arguments, two more arguments *rec_l* and *rec_r* are needed. Those variables in the elimination rule beginning with *rec_* are recurse variables and are evaluated to recurse down the associated recursive introduction variable, e.g., evaluating *rec_l* would recurse down the tree bound to l. The arguments to the recurse variables are specified by the computation rules.

The recurse variables are used in a functional program at the point where the function should be applied to an argument of the data constructor. In a strongly-typed language this only makes sense if that parameter was specified in the introduction rule to be of the same type as the data constructor. Thus, recurse variables only occur with recursive introduction variables.

For example, a function to determine if a specified element is in a tree can be defined as:

$$\begin{aligned} \textit{tree-search} \equiv \lambda x.\lambda ele.\textit{BinTree-elim}(x, \lambda a.(ele \text{ EQUAL } a), \\ \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r)) \end{aligned}$$

SPIDER contains pattern-directed function definitions to make this easier for the application programmer to define. It also contains a "recurse" form so that the recurse variables *rec_l* and *rec_r* are not specified directly by the application program but by reference to their associated recursive introduction variables. The SPIDER definition of `tree-search` is:

```
defsfun tree-search BinTree(A)
     (?ele)
  leaf(?a) => ?a equal ?ele
  node(?l,?r) => recurse(?l) or recurse(?r)
```

which has the type description:

$$\textit{tree-search} : BinTree(A) \times A \rightarrow Boolean$$

It could be used as follows:

```
>> tree-search(node(leaf(a),leaf(b)),b)
true
>> tree-search(node(leaf(a),node(leaf(b),leaf(c))),d)
false
>> tree-search(node(leaf(a),node(leaf(b),leaf(c))),b)
true
```

Computation Rules: *Computation rules* tell how a specific *BinTree-elim* instance should be evaluated. There is a computation rule for each data constructor which is used when x matches that data constructor (i.e., x is either **leaf** or **node**$(\_,\_)$). This is sufficient because all members of the type must have been formed through some composition of data constructors. Since BinTree has two data constructors, there are two computation rules. They are of the form

$$\textit{BinTree-elim}(<constructor>, leaf\_abs, node\_abs) = <value> .$$

This equation holds in the class of canonical expressions generated by the constructor. Thus the full conclusion is

$$\textit{BinTree-elim}(<constructor>, leaf\_abs, node\_abs) = <value> \in C[<constructor>].$$

The computation rules for $BinTree(A)$ are:

leaf-computation

$$\frac{\begin{array}{l} [\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\ a \in A \\ [\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]]\!] \\ [\![\ l \in BinTree(A) \\ \quad r \in BinTree(A) \\ \quad rec\_l \in C[l] \\ \quad rec\_r \in C[r] \\ \quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] \\ ]\!] \end{array}}{\textit{BinTree-elim}(\mathrm{leaf}(a), leaf\_abs, node\_abs) = leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]}$$

If x is **leaf(_)**, then the expression $\mathit{BinTree\text{-}elim}(x, \mathit{leaf\_abs}, \mathit{node\_abs})$ will have value $\mathit{leaf\_abs}$.

node-computation

$$
\begin{array}{l}
[\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\
l \in BinTree(A) \\
r \in BinTree(A) \\
[\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]\,]\!] \\
[\![\ l \in BinTree(A) \\
\quad r \in BinTree(A) \\
\quad rec\_l \in C[l] \\
\quad rec\_r \in C[r] \\
\quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] \\
]\!] \\
\hline
\mathit{BinTree\text{-}elim}(\mathrm{node}(l, r), leaf\_abs, node\_abs) \\
\qquad\qquad = \ node\_abs(l, r, \mathit{BinTree\text{-}elim}(l, leaf\_abs, node\_abs), \\
\qquad\qquad\qquad\qquad\qquad \mathit{BinTree\text{-}elim}(r, leaf\_abs, node\_abs)) \\
\quad \in \ C[\mathrm{node}(l, r)]
\end{array}
$$

If x is a *node*, the specified *node_abs* function will be evaluated over the arguments to *node* and a recursive *BinTree-elim* call. This rule defines how recursion is to take place over the arguments of the recursive data constructor *node*.

The premises of a computation rule are very easy to calculate from the elimination rule. The major premise of the elimination rule is replaced with the premises of the corresponding introduction rule. It is the conclusion of the computation rule which requires work to calculate. Thus for brevity (and clarity), we will often omit the premises of the computation rules and just give their conclusions, yielding:

leaf-computation

$$
\begin{array}{l}
\hline
\mathit{BinTree\text{-}elim}(\mathrm{leaf}(a), leaf\_abs, node\_abs) \ = \ leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]
\end{array}
$$

node-computation

$$
\begin{array}{l}
\hline
\mathit{BinTree\text{-}elim}(\mathrm{node}(l, r), leaf\_abs, node\_abs) \\
\qquad\qquad = \ node\_abs(l, r, \mathit{BinTree\text{-}elim}(l, leaf\_abs, node\_abs), \\
\qquad\qquad\qquad\qquad\qquad \mathit{BinTree\text{-}elim}(r, leaf\_abs, node\_abs)) \\
\quad \in \ C[\mathrm{node}(l, r)]
\end{array}
$$

Rather than describe the rules in terms of their proof capabilities (see [ML82, BCM88, Bac86a, BC85]), we give an operational description in terms of SPIDER and WEB. The elimination rule tells how to construct the functional *BinTree-elim* which has three parameters. The first argument must be of type BINTREE(A), the second a lambda abstraction with one parameter, and the third a lambda abstraction with four parameters.

User-defined functions in SPIDER are specified by giving function definitions (lambda abstractions) for *leaf_abs* and *node_abs*, which have arguments as defined in the BinTree-elimination rule. This user-defined function is called by giving it a reference in the knowledge base. For BinTree(A), this must be either a *leaf* or a *node*. The appropriate computation rule is chosen, and the abstraction *leaf_abs* or *node_abs* is lazily evaluated.

Theoretically, SPIDER functions can only be called recursively using recurse variables. The system can enforce this but does not in the current implementation because there is no reason for a SPIDER program to violate this constraint. When this is enforced, it can be shown that functions which are *locally terminating* are also *globally terminating*. Basically, this means if all recursive calls were replaced with a appropriately typed "stub", then if this modified function halts, i.e. is locally terminating, then the original function halts. This can be made rigorous.

Now, we can explain the class construct $C[w]$. The *type premise* in the elimination rule states $[\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!]$. The elements in $C[w]$ are the canonical elements generated by the type using the the elimination rule. Notice that the conclusion of the elimination rule $\textit{BinTree-elim}(x, leaf\_abs, node\_abs) \in C[x]$ is specifying what all the elements of the class are. $C[x]$ is a stratified class where the elements of the class are types indexed by the elements of BinTree(A).

Looking at the class in the context of the `tree-search` function, because *tree-search* is of type $BinTree(A) \times A \rightarrow Boolean$, the class construct contains (at most) the two elements *true* and *false*. Thus the infinite collection of types can be grouped into two strata: those types that have *true* as its only element and those that have *false* as its only element. Each type expression has one and only one canonical element because the computations halt and are deterministic. The computation rules specify how to group the type expressions into the strata.

2.2.2 Simple Types

Now, we look at a simple, nonrecursive type to show how it would be defined using constructive type theory. The BOOLEAN type consists of two nullary data constructors *true()* and *false()*. It inference rules are:

Boolean-formation

$$\frac{}{\textit{Boolean type}}$$

true-introduction

$$\frac{}{\text{true} \in \textit{Boolean}}$$

false-introduction

$$\frac{}{\text{false} \in \textit{Boolean}}$$

Boolean-elimination

$$\frac{\begin{array}{ll} [\![w \in \textit{Boolean} \quad \triangleright \quad C[w]\ \textit{type}]\!] & \text{— type premise} \\ x \in \textit{Boolean} & \text{— major premise} \\ \textit{true\_val} \in C[\text{true}] & \text{— true-premise} \\ \textit{false\_val} \in C[\text{false}] & \text{— false-premise} \end{array}}{\textit{Boolean-elim}(x, \textit{true\_val}, \textit{false\_val}) \ \in \ C[x]}$$

Note that *Boolean-elim* is identical to what we normally consider to be the "if" function. We will sometimes use "if" for *Boolean-elim* in SPIDER programs for clarity.

true-computation

$$\frac{\begin{array}{l} [\![w \in \textit{Boolean} \quad \triangleright \quad C[w]\ \textit{type}]\!] \\ \textit{true\_val} \in C[\text{true}] \\ \textit{false\_val} \in C[\text{false}] \end{array}}{\textit{Boolean-elim}(\text{true}, \textit{true\_val}, \textit{false\_val}) \ = \ \textit{true\_val} \ \in C[\text{true}]}$$

false-computation

$$\frac{\begin{array}{l} [\![w \in \textit{Boolean} \quad \triangleright \quad C[w]\ \textit{type}]\!] \\ \textit{true\_val} \in C[\text{true}] \\ \textit{false\_val} \in C[\text{false}] \end{array}}{\textit{Boolean-elim}(\text{false}, \textit{true\_val}, \textit{false\_val}) \ = \ \textit{false\_val} \ \in C[\text{false}]}$$

2.3 Knowledge Models

Knowledge models are formalized as an algebra over the types defined by constructive type theory and their methods. The same approach has been used to define modules in the programming language SML, but we show it is also useful for knowledge bases. Algebraic methods have been used to specify data models, e.g., relational algebra, and is used here as a technique for specifying new data models in a knowledge base.

We demonstrate this approach on two representation schemes. $\mathcal{ALRC}$ [Sch89a] is a formal language which captures the key constructs of term subsumption languages such as KL-ONE [BS85]. Situation theory [BE90a] is a mechanism for representing natural language semantics.

2.3.1 $\mathcal{ALRC}$

Terminological subsumption languages were developed to automatically create hierarchies where the concepts are defined as terms in some language (initially KL-ONE), and the hierarchy shows the subsumption relations between the terms. The process of placing new terms into the hierarchy is called *classification*.

In order to develop a terminological reasoner, the type constructors CONCEPT(A) and ROLE(A) are defined along with the data constructors `new-prim-concept`, `new-concept`, and `new-role`, which store the new constructs in the knowledge base along with their definition (or restriction, if any). The methods `defprimconcept`, `defconcept`, `defrole` use other methods to find the correct location of the concept/role in its hierarchy (classification) and then use one of the data constructors to store it. The restrictors and combinators (e.g., and, or, not, all, some) are defined as data constructors on CONCEPT(A) while other reasoners (such as subsumption and classification) are defined as methods.

The type constructors are given with the type description of their data constructors. The constructive type theory inference rules are fairly straightforward and are omitted.

Roles are multi-valued features of a concept. Roles are defined by giving them a name.

$Role(A)$

$\quad new\text{-}role : \quad A$

Concepts are formed by creating new primitive concepts or creating relations between existing concepts and/or roles. If C and D are concepts, R is a role, and P and Q are list of roles, the relations in $\mathcal{ALRC}$ can be defined as:

$$
\begin{aligned}
&\textit{and}: \quad C \sqcap D \\
&\textit{or}: \quad C \sqcup D \\
&\textit{not}: \quad \neg C \\
&\textit{all}: \quad \forall R{:}\, C \\
&\textit{exists}: \quad \exists R{:}\, C \\
&\textit{equal}: \quad P = Q
\end{aligned}
$$

These can be defined as data constructors for the CONCEPT(A) type constructor. Primitive concepts are defined by giving them a name and other concepts are defined by associating a name with a concept. This recursive definition combines concept definition (*new-prim-concept*, *new-concept*), concept forming operators (*and*, *or*, *not*), role restrictors (*all*, *exists*) and simple role value maps (*equal*).

$$
\begin{aligned}
&\mathit{Concept}(A) \\
&\qquad \textit{new-prim-concept}: \quad A \\
&\qquad \textit{new-concept}: \quad A \times \mathit{Concept}(A) \\
&\qquad \textit{and}: \quad \mathit{Concept}(A) \times \mathit{Concept}(A) \\
&\qquad \textit{or}: \quad \mathit{Concept}(A) \times \mathit{Concept}(A) \\
&\qquad \textit{not}: \quad \mathit{Concept}(A) \\
&\qquad \textit{all}: \quad \mathit{Role}(A) \times \mathit{Concept}(A) \\
&\qquad \textit{exists}: \quad \mathit{Role}(A) \times \mathit{Concept}(A) \\
&\qquad \textit{equal}: \quad \mathit{List}(\mathit{Role}(A)) \times \mathit{List}(\mathit{Role}(A))
\end{aligned}
$$

We can define these type constructors CONCEPT(A) and ROLE(A) over SYMBOL to create two abstract data types CONCEPT(SYMBOL) and ROLE(SYMBOL). These types (instantiated type constructors) form a database schema which has a signature of:

$Role(Symbol)$

$$new\text{-}role: \ Symbol \rightarrow Role(Symbol)$$

$Concept(Symbol)$

$$new\text{-}prim\text{-}concept: \ Symbol \rightarrow Concept(Symbol)$$

$$new\text{-}concept: \ Symbol \times Concept(Symbol) \rightarrow Concept(Symbol)$$

$$and: \ Concept(Symbol) \times Concept(Symbol) \rightarrow Concept(Symbol)$$

$$or: \ Concept(Symbol) \times Concept(Symbol) \rightarrow Concept(Symbol)$$

$$not: \ Concept(Symbol) \rightarrow Concept(Symbol)$$

$$all: \ Role(Symbol) \times Concept(Symbol) \rightarrow Concept(Symbol)$$

$$exists: \ Role(Symbol) \times Concept(Symbol) \rightarrow Concept(Symbol)$$

$$equal: \ List(Role(Symbol)) \times List(Role(Symbol)) \rightarrow Concept(Symbol)$$

This schema can be used to define a simple terminology with concepts *person*, *male*, and *female* and with roles *child* and *sex* as follows:

```
new-prim-concept(person)
new-prim-concept(male)
new-concept(female, not(male))
new-prim-role(child)
new-prim-role(sex)
```

The additional concepts *parent*, *father*, and *mother* can be added and classification can be used to keep track of the subsumption relations. These are defined as:

```
new-concept(parent, and(person,
                        and(exists(child, person),
                            all(child, person))))
new-concept(father, and(parent, male))
new-concept(mother, and(parent, female))
```

The signature is then extended by the methods defined on the type. For example, classification is a function from a concept to a set of concepts. Constraints on the schema are formalized as axioms on the algebra are defined by the body of the methods.

The advantages over a dedicated terminological reasoner occur because of the extensibility of SPIDER and uniform storage of the knowledge in WEB. For example:

1. A more expressive terminological language can be developed as an extension to this one. The reasoning methods can then be defined so that the (theoretically) most efficient reasoner will be used when possible. For example, adding attributes and n-ary roles will not restrict the existing representation (only extend it), and appropriate extensions to the reasoning methods would allow for more tractable (decidable) [Sch89a] or expressive reasoners [Sch89b], respectively.
2. WEB allows for cyclic feature structures [Car92]. Thus, the KL-ONE style type definitions can be extended to include *circular definitions* (terminological cycles) [NS90].

2.3.2 Situation Theory

Situation theory [BE90a] is a theory of information content which has been applied to problems in logic, linguistics and databases. It supports general, heterogeneous inferencing through partial information built up from infons. Infons are pieces of information. The version of situation theory we use here requires that six type constructors be created. They are given informally here, along with the type description of their data constructors. The constructive type theory inference rules for the types are simple and are omitted.

$Relation(R)$

$\quad rel: \quad R$

An infon is a piece of information and consists of a relation applied to a sequence of objects in the domain. The type definition for SEQUENCE(A) is the same as LIST(A), which is given in appendix C. Infons are defined in situation theory as $\langle\langle Relation, a_1, a_2, \ldots, a_n; i\rangle\rangle$ where *Relation* is the relation of the infon, $a_1, a_2, \ldots, a_n$ are the arguments to the relation, and $i = 1$ if the relation holds and $i = 0$ if the relation does not hold. We use the data constructors *positive* and *negative* to specify whether the relationship holds or does not hold, respectively.

$Infon(R, A)$

$\quad positive: \quad Relation(R) \times Sequence(A)$

$\quad negative: \quad Relation(R) \times Sequence(A)$

Objects are the elements in the domain.

$Object(A)$

$obj: \;\; A$

Infons can be combined into a lattice to form complex infons. For example, to express the complex infon "Joe owns either a Chevy or a Ford" in situation theory, a join is used:

$$\langle\!\langle \mathit{Owns}, \mathit{Joe}, \mathit{Chevy}; 1 \rangle\!\rangle \;\vee\; \langle\!\langle \mathit{Owns}, \mathit{Joe}, \mathit{Ford}; 1 \rangle\!\rangle$$

This is defined using the data constructors over SYMBOL as:

```
owns == rel(Owns)
joe == obj(Joe)
join(positive(owns, joe, obj(Chevy)),
     positive(owns, joe, obj(Ford)))
```

Other complex infons are defined similarly.

$ComplexInfon(R, A)$

$top : ()$

$bottom : ()$

$meet : \;\; Infon(R, A) \times Infon(R, A)$

$join : \;\; Infon(R, A) \times Infon(R, A)$

Situations are sets of complex infons over objects.

$Situation(A)$

$sit : \;\; Set(ComplexInfon(A, Object(A)))$

We can instantiate the type constructor using the type SYMBOL for the type variable A. This creates the abstract data type SITUATION(SYMBOL). All the abstract data types defined for situation theory form a schema which yields an algebra with signature shown in Figure 3.

$Situation(Symbol)$

$$sit: \ \ Set(ComplexInfon(Symbol, Object(Symbol))) \rightarrow Situation(Symbol)$$

$$ele: \ \ MVA(ComplexInfon(Symbol, Object(Symbol))) \times Id \rightarrow Set(ComplexInfon(Symbol, Object(Symbol)))$$

$$empty\text{-}set: \ \ () \rightarrow Set(ComplexInfon(Symbol, Object(Symbol)))$$

$$top: () \rightarrow ComplexInfon(Symbol, Object(Symbol))$$

$$bottom: () \rightarrow ComplexInfon(Symbol, Object(Symbol))$$

$$meet: \ \ Infon(Symbol, Object(Symbol)) \times Infon(Symbol, Object(Symbol)) \rightarrow ComplexInfon(Symbol, Object(Symbol))$$

$$join: \ \ Infon(Symbol, Object(Symbol)) \times Infon(Symbol, Object(Symbol)) \rightarrow ComplexInfon(Symbol, Object(Symbol))$$

$$obj: \ \ Symbol \rightarrow Object(Symbol)$$

$$rel: \ \ Symbol \rightarrow Relation(Symbol)$$

$$positive: \ \ Relation(Symbol) \times Sequence(Object(Symbol)) \rightarrow Infon(Symbol, Object(Symbol))$$

$$negative: \ \ Relation(Symbol) \times Sequence(Object(Symbol)) \rightarrow Infon(Symbol, Object(Symbol))$$

Figure 3: Signature for Situation(Symbol)

Chapter 3
Graph Logic

Simple things do not differ from one another by added differentiating factors as composites do.
— *Thomas Aquinas*

Graphs are a natural way of representing many kinds of information. Graphs are frequently used to explain ideas, organizations, problems, and solutions. They are also useful for describing data structures and knowledge representation schemes. We introduce a simple logic for formalizing graphs, and then we implement that logic as the logic programming language WEB.

As graphical workstations are becoming more prevalent and people are beginning to appreciate their usefulness in presenting information, it is becoming more important to investigate the possibilities in reasoning with graphical representations. Work is already being done in programming using iconic representations, and this can be extended to having the programs themselves be graphs. One step toward this goal is to formalize graphs in such a manner which allows inferencing to be made of graphical structures. This is best done in terms of a logic. Another advantage of this approach is that the declarative paradigm lends itself to treating programs as data. This is a fascinating possibility, but we do not pursue it here and require that the graph structure be accessed through the type system of SPIDER.

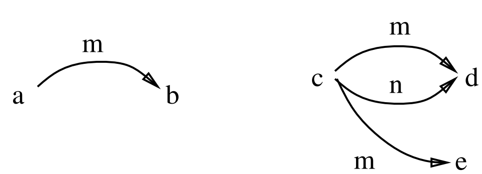
This chapter contains the details of WEB. We give the graph querying algorithm with examples to explain its use in section 3.1. We also formalize WEB in terms of a graph logic in section 3.2. We give a definition for labeled graphs in terms of vertices, edges, and labels and show how the graphs are built using the logic incorporated in WEB. In section 3.3, we give an overview of the persistent knowledge store which forms the fourth (lowest) level in WEAVE's architecture.

3.1 Graph Querying Algorithm

Graph querying is defined in WEB by the function `query`. The function `query` is best understood as a series of computations where a set of constraints holds between each adjacent computation. The series begins with $\langle \sigma, B_0 \rangle$ where σ is a sequence in graph logic and B_0 is an empty binding table. A *sequence* is an ordered collection of binary predicates and is more rigorously defined in section 3.2. A *binding table* consists of a set of binding entries where each *binding entry* is a set of tuples.

3.1.1 Initial Example

For example, a query of the one-element sequence $(m\ c\ ?x)$ against the graph



defined by the set of binary predicates, called *triads*:

$(m\ a\ b)$
$(m\ c\ d)$
$(n\ c\ d)$
$(m\ c\ e)$

would result in a binding table with one entry:

?x
d
e

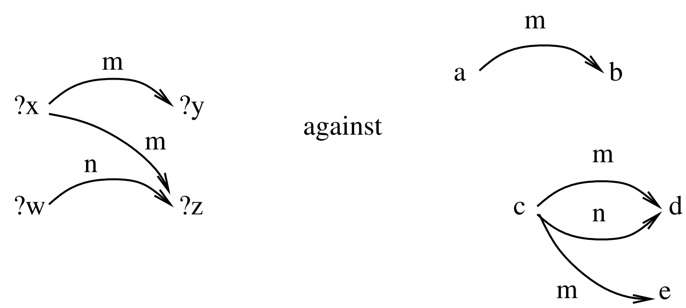
because $(m\ c\ ?x)$ matches against the $(m\ c\ d)$ and $(m\ c\ e)$ triads.

The query $(m\ ?x\ ?y); (n\ ?w\ ?z)$ against the same graph would result in a binding table with two binding entries, where the sequence $\tau_1; \tau_2$ denotes that the triads τ_1 and τ_2 must match with nonconflicting variable bindings.

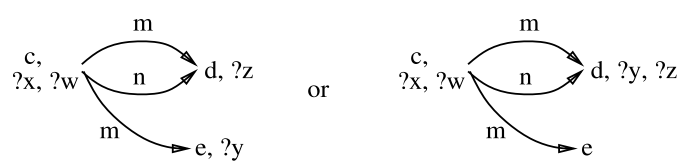
?x	?y
a	b
c	d
c	e

?w	?z
c	d

Now consider what should happen if the query also contains the triad $(m\ ?x\ ?z)$, ie, it is $(m\ ?x\ ?y); (n\ ?w\ ?z); (m\ ?x\ ?z)$. We then want to unify the query with the WEB graph, i.e., query



This would result in



which would be described by the following binding table (the order of the columns or rows is irrelevant).

?w	?x	?y	?z
c	c	d	d
c	c	e	d

3.1.2 Specification of Cases

We now define the graph query algorithm in terms of constraints on the binding tables which hold as each triad in a sequence is processed. The algorithm uses a function *llquery* from the persistent knowledge store which returns all triads in the knowledge base which match the parameterized triad given as an argument.

If σ is a sequence with $|\sigma| = n$, consider the computation series $B_0 \sigma_1 B_1 \sigma_2 B_2 \cdots \sigma_n B_n$ where σ_i is the i^{th} triad in σ, and the final result (binding table) is given by B_n. The following are true of each computation $B_{i-1} \sigma_i B_i$:

1) If $B_{i-1} =$ *:FAIL*,

   then $B_i =$ *:FAIL*, thus $B_i, \ldots, B_n =$ *:FAIL* and $query(\sigma) =$ *:FAIL*.

2) If σ_i has *no* variables,

a) If the result of σ_i is empty, i.e., $llquery(\sigma_i) = \emptyset$,

then the query fails, i.e., $B_i, \ldots, B_n =$ *:FAIL*.

b) If $llquery(\sigma_i) \neq \emptyset$,

then $B_i = B_{i-1}$, i.e, the binding table remains unchanged.

3) If σ_i has one or two variables[4] and $llquery(\sigma_i) = \emptyset$,

then the query fails.

4) If σ_i has one variable which *does not* already occur in B_{i-1}, call it x,

then

$$B_i = B_{i-1} \cup \{ \langle x, llquery(\sigma_i)_x \rangle \}$$

where $llquery(\sigma_i)_x$ denotes the values specified by $llquery(\sigma_i)$ for the variable x, and $\langle x, \{t_1, t_2, \ldots\} \rangle$ is the linear notation for a binding entry for x with values $\{t_1, t_2, \ldots\}$.

5) If σ_i has one variable which *does* already occur in B_{i-1}, call it x,

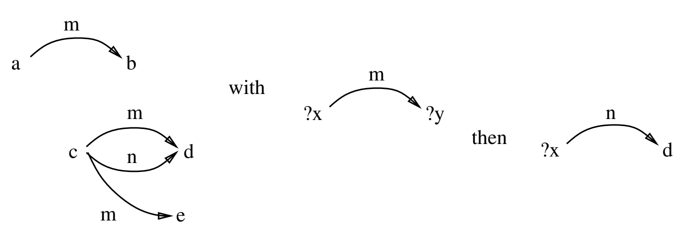
then

$$B_i = (B_{i-1} - B^x_{i-1}) \cup B^x_{i-1} |_{x \in llquery(\sigma_i)_x}$$

where B^x_{i-1} denotes the binding entry for x (along with the binding for any other variables in that entry). The notation $\langle$*binding entry*$\rangle |_{\langle restriction \rangle}$ is a selection operation — select tuples from B^x_{i-1} where $x \in llquery(\sigma_i)_x$.

Example

Consider the query $(m\ ?x\ ?y); (n\ ?x\ d)$ against the same graph as above (p. 39). This is described by the computation series $B_0\ (m\ ?x\ ?y)\ B_1\ (n\ ?x\ d)\ B_2$. We unify



The first unification yields the binding table B_1 with one binding entry

?x	?y
a	b
c	d
c	e

[4] A query is not allowed to have three variables, because this would simply return the entire knowledge base. This restriction is removed in the implementation under certain useful conditions to further constrain the binding tables.

$B_1^{?x}$ selects that binding entry. Since $llquery(n, ?x, d) = (n\ c\ d)$, we select from $B_1^{?x}$ the tuples where the value of $?x$ is a member of $llquery(n, ?x, d)_{?x} = \{c\}$. Thus,

$$B_1^{?x}\,|_{?x \in llquery(n,?x,d)_{?x}} \;=\; \begin{array}{|c|c|}\hline ?\mathrm{x} & ?\mathrm{y} \\ \hline \mathrm{c} & \mathrm{d} \\ \mathrm{c} & \mathrm{e} \\ \hline \end{array}$$

which is the only binding entry in B_2 (the final result).

6) If σ_i has two variables, say x and y, *neither* of which occur in B_{i-1},
then

$$B_i = B_{i-1} \ \cup \ \{ \ \langle xy, llquery(\sigma_i)_{xy} \rangle \ \}$$

Example

As in the first step of the example immediately above where

$$B_1 = B_0 \ \cup \ \{ \ \langle ?x?y, \{(a\ b), (c\ d), (c\ e)\} \rangle \ \}$$

7) If σ_i has two variables, *one* of which occurs in B_{i-1}, say x, and the other does not, say y,
then

$$B_i = (\ B_{i-1} \ - \ B_{i-1}^x \ ) \ \cup \ \Big[\big(B_{i-1}^x \,|_{x \in llquery(\sigma_i)_x} \big) \ \times \ \big(\langle xy, llquery(\sigma_i)_{xy} \rangle \,|_{x \in B_{i-1}^x} \big) \Big]$$

The new binding entry is the same as the old one cross the *llquery* result where the x's occur in both the old binding entry and the query. This can be written as a natural join:

$$B_i = (\ B_{i-1} \ - \ B_{i-1}^x \ ) \ \cup \ (\ B_{i-1}^x \ \bowtie \ \langle xy, llquery(\sigma_i)_{xy} \rangle \ )$$

8) If σ_i has two variables, say x and y, *both* of which occur in B_{i-1}, and they are in the *same* binding entry, i.e., $B_{i-1}^x = B_{i-1}^y$,
then

$$B_i = B_{i-1}^{xy} \,|_{(x\ y) \in llquery(\sigma_i)_{xy}}$$

9) If σ_i has two variables, say x and y, *each* of which occur in B_{i-1}, but in *separate* binding entries, i.e., $B_{i-1}^x \neq B_{i-1}^y$,
then

$$B_i = (\ B_{i-1} \ - \ B_{i-1}^x \ - \ B_{i-1}^y \ ) \ \cup \ \big((\ B_{i-1}^x \ \bowtie \ \langle xy, llquery(\sigma_i)_{xy} \rangle \ ) \ \bowtie \ B_{i-1}^y \big)$$

which is equivalent to

$$B_i = (\ B_{i-1} \ - \ B_{i-1}^x \ - \ B_{i-1}^y \ ) \ \cup \ \big((\ B_{i-1}^x \ \times \ B_{i-1}^y \ ) \,|_{(x\ y) \in \langle xy, llquery(\sigma_i)_{xy} \rangle} \big)$$

Example

See above (p. 40) for the query $(m\ ?x\ ?y); (n\ ?w\ ?z); (m\ ?x\ ?z)$.

3.1.3 Algorithm Complexity

Although the emphasis of this work is on design, it is also important to give a rough estimate of run-time efficiency. The most compute intensive aspect of WEAVE is the graph querying algorithm. In part, this occurs because of the expressive nature of higher-order cyclic graphs, but it also occurs because the implemented algorithm was developed to closely match the theoretical properties defined in the specification of cases above.

Worst case performance occurs when the graph constructor is accessing a binary tree of totally connected graphs (cliques). A cartesian product must be formed between each of the members of the cliques. This requires exponential space to calculate the full relation in terms of the queried sequence. However, this is linear in terms of the result, and thus if the user wants the full relation, this is the best any system can do. It is possible to get worst case performance in terms of the result which requires exponential space by calculating the full relation, then throwing it away. However, reordering (optimizing) the query sequence will eliminate this undesired performance and result in $\mathcal{O}(n^3)$ performance, where n is the length of the query sequence. This occurs with a highly interconnected graph formed from the complement with respect to the knowledge base of a set of almost totally connected graphs, but this performance is linear with respect to the number of triads in the knowledge base. It is not known if there is a query which is worse than linear for both knowledge base size and query result.

These worst case graph constructors are fairly unrealistic, but they do demonstrate that poorer performance should be expected with highly interconnected graphs. When optimizing queries it is usually useful to place the triads in the query sequence which match the most triads in the knowledge base at the end of the sequence.

3.2 Formalization of WEB

The logical formalism of WEB is used to build and query against graph structures. In this section, we give a standard definition for graphs and show how they are built using WEB.

3.2.1 Definitions

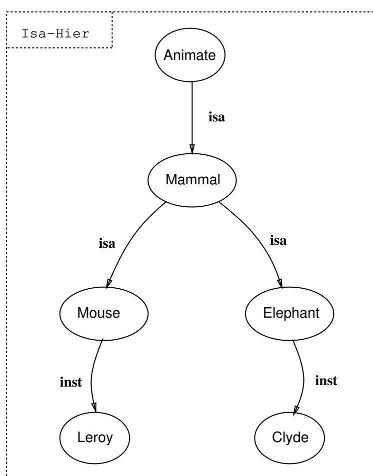
WEB consists of three primary structures: permissions, links, and nodes. Permissions are binary predicates over links, and they are partially encapsulated by nodes. An alternative nomenclature might be predicates, atoms, and context/theory/microtheory/ontology. A more graphical nomenclature would be arcs, vertices, and subwebs. We use this terminology because it can make more complex examples clearer, though it may make the simple ones more confusing.

For example, the `Isa-Hier` *node* might contain:

```
(inst Leroy Mouse)
(inst Clyde Elephant)
(isa Mouse Mammal)
(isa Elephant Mammal)
(isa Mammal Animate)
```

`inst` and `isa` are *permissions* while `Leroy`, `Mouse`, `Clyde`, `Elephant`, `Mammal`, and `Animate` are *links*.

Graphically, this can be represented as:



3.2.1.1 Definition of WEB Primitives

A *node* is a tuple $\langle m, \mathcal{L}, \mathcal{P}, \mathcal{N} \rangle$

- m — the name of the node.
- $\mathcal{L}$ — a collection of links which are immediately defined and encapsulated by the node.
- $\mathcal{P}$ — a collection of permissions which are immediately defined and encapsulated by the node.
- $\mathcal{N}$ — a collection of nodes which are immediately defined and encapsulated by the node.

The *encapsulation hierarchy* Θ_{TOP} is a node named TOP which contains all other nodes. These may or may not be arranged in a hierarchical manner, but that will not effect the theoretical results. We now collect all the nodes, permissions, and links defined in the encapsulation hierarchy.

We say

- ∘ Θ_{node} denotes all nodes in all nodes in Θ_{TOP}.
- ∘ Θ_{perm} denotes all permissions in all nodes in Θ_{TOP}.
- ∘ Θ_{link} denotes all links in all nodes in Θ_{TOP}.

A *triad*, τ, is a 3-tuple $\langle p, s, d \rangle$ where $p \in \Theta_{perm}$ and $s, d \in \Theta_{link}$ (read as "source" and "destination").

A *knowledge base* is defined by the pair $\langle \Theta_{\mathrm{TOP}}, \mathcal{T} \rangle$

- Θ_{TOP} — the encapsulation hierarchy
- $\mathcal{T}$ — a collection of triads (instantiated binary predicates)

A *sequence*, σ, is an ordered collection of triads.

If τ is the i^{th} triad in σ, we write $\sigma(i) = \tau$.

We write $\tau \in \sigma$ or $\langle p, s, d \rangle \in \sigma$ if the triad occurs somewhere in the sequence σ.

A *WEB structure*, w, is a pair (q, σ) where q is a link in some triad in the sequence, i.e., for some $\langle p, s, d \rangle \in \sigma$ either $q = s$ or $q = d$. We call q the *distinguished link* of the WEB structure.

We occasionally write q_w and σ_w to refer to the distinguished link and sequence of w, respectively.

A WEB structure, w, is *trivial* if $\sigma_w = \emptyset$.

A *simple WEB constructor*, c, is a function $c \colon \Theta^{*}_{link} \to w$ where w is a WEB structure. All links in the arguments to c must occur in the links of σ_w.

An *edge-labeled graph* $G = (V, E, L)$ consists of a set of vertices V, edges E, and labels L, where $E \subseteq L \times V \times V$. This is a directed, possibly cyclic graph which allows for multiple arcs between vertices, if the arcs have different labels.

The WEB graph building function $\textsc{Build}[\![\ ]\!]$ creates a graph from a WEB sequence. To define $\textsc{Build}[\![\ ]\!]$, we show its actions on the vertex, edge, and label views of a graph $V[G]$, $E[G]$, and $L[G]$.

$$\textsc{Build}[\![(p\ s\ d);\sigma]\!]:$$

$$V[G] = V[G'] \ \cup \ \{s, d\}$$

$$E[G] = E[G'] \ \cup \ \{(p, s, d)\}$$

$$L[G] = L[G'] \ \cup \ \{p\}$$

where $G' = \textsc{Build}[\![\sigma]\!]$.

The empty sequence $\sigma_\emptyset$ builds an empty graph $\textsc{Build}[\![\sigma_\emptyset]\!] = (\emptyset\ \emptyset\ \emptyset)$.

Theorem $\textsc{Build}[\![\ ]\!]$ is invariant under sequence permutation.
Proof:

Because a graph is a collection of three sets, it does not matter what order the elements are put into the sets. ■

Two graphs are isomorphic if they are identical except possibly for link names.

Two sequences are *isomorphic* if the graphs they build are isomorphic.

Two sequences are *equivalent* if they contain the same triads (not necessarily in the same order).

Proposition If two sequences are equivalent, they are also isomorphic.

Theorem Two sequences are equivalent iff they build identical graphs.
Proof:

Because $\textsc{Build}[\![\ ]\!]$ is invariant under sequence permutation, equivalent sequences will build identical graphs. ■

Theorem Graph querying is invariant under sequence permutation.
Proof:

If a sequence σ and its permutation σ' build identical graphs, then parameterizations of them σ_v and σ'_v will yield identical results on identical bindings. Because triad querying with *llquery* is independent of previous triad queries, the order of the sequence does not matter. ■

The importance of this theorem in a practical system is that we can reorder the triads in a sequence to optimize performance of the graph query algorithm.

If two sequences are equivalent, they are in the same equivalence class, written $[\sigma]$.

A sequence σ_1 is a *subsequence* of σ_2, written $\sigma_1 \sqsubseteq \sigma_2$, if $\{\tau \mid \tau \in \sigma_1\} \subseteq \{\tau \mid \tau \in \sigma_2\}$.

A *sort* s_w is defined by a WEB structure, written as $w = (q_s, \sigma_s)$, and contains the WEB structures as

$$s_w = \{(q_s, \sigma) \mid \sigma_s \sqsubseteq \sigma\}$$

thus s_w contains all WEB structures which are more general than w. Note the set $\{(q_s, \sigma) \mid \sigma \in [\sigma_s]\}$ is a subset of $\{(q_s, \sigma) \mid \sigma_s \sqsubseteq \sigma\}$.

We say a sequence, σ, is an element of a sort $s_{(q_w,\sigma_w)}$ if (i) for some $\langle p, s, d\rangle \in \sigma, s = q_w$ *or* $d = q_w$ and (ii) $(q_w, \sigma) \in s_{(q_w,\sigma_w)}$.

Two sorts s_{w_1}, s_{w_2} are *equivalent* if there is an isomorphic mapping between all the links of s_{w_1} and s_{w_2} that holds for all WEB structures in each sort.

Proposition If two sorts have equivalent defining sequences, the two sorts are equivalent.

Theorem Two sorts are equivalent iff their defining sequences are isomorphic.
Proof:

From the proposition above, and if there is an isomorphic mapping of all links, then the graphs must be isomorphic. ■

Two sorts s_{w_1}, s_{w_2} are *isomorphic* if their defining graphs are isomorphic and preserve the distinguished link.

Lemma When for any two binding of variables, a constructor returns two WEB structures w_1 and w_2, the sorts s_{w_1} and s_{w_2} are equivalent.

A graph is *well-founded with respect to* a set of links and a set of constructors when there is some binding of links as arguments of the constructors which would build the graph.

If a graph is well-founded with respect to some Θ_{link} and Θ_{cons}, we say it is *well-founded.*

Theorem All finite WEB graphs are well-founded.
Proof:

Let Θ_{link} contain all the links in the graph and Θ_{cons} contain one constructor whose defining sequence builds each triad in the graph. This is possible to construct when there is a finite number of triads in the graph. ■

3.2.1.2 Definition of SPIDER Types

A SPIDER type constructor is defined as a collection of SPIDER data constructors. The SPIDER data constructor is a typed, n-ary SPIDER function which is associated with a n-ary WEB constructor (of the same arity). Each data constructor parameter is typed with a SPIDER type. A SPIDER type is an instantiated type constructor. A polymorphic SPIDER type is a SPIDER type which contains a type variable [CW85].

Proposition A WEB graph can be of several SPIDER types.

3.2.2 Structure Checking

Two sequences σ_1, σ_2 *overlap* if for some $\tau_1 \in \sigma_1$ and $\tau_2 \in \sigma_2$, $\tau_1 = \tau_2$.

Two WEB structures (q_1, σ_1) and (q_2, σ_2) *overlap* if $q_1 = q_2$ and σ_1 and σ_2 overlap.

Proposition If (q_1, σ_1) and (q_2, σ_2) overlap, there exists a nonempty sequence $\overline{\sigma}$ such that $\overline{\sigma} \sqsubseteq \sigma_1$ and $\overline{\sigma} \sqsubseteq \sigma_2$.

Proposition If WEB structures w_1 and w_2 overlap, there exists a sort which is a superset of both s_{w_1} and s_{w_2}.

Theorem If w_1 and w_2 overlap, there exists a *unique* sort which is a superset of both s_{w_1} and s_{w_2} and which is maximally specific.

Proof:

It is $s_{(q_w, \overline{\sigma})}$ for most specific $\overline{\sigma}$ such that $\overline{\sigma} \sqsubseteq \sigma_1$ and $\overline{\sigma} \sqsubseteq \sigma_2$. ■

Corollary These unique sorts form a complete lattice on set inclusion for any finite set of WEB structures.

Two WEB constructors c_1, c_2 overlap if their defining WEB structures overlap (with variable renaming).

Theorem For any finite collection of WEB constructors, the equivalence classes on their sorts form a meet semi-lattice over the subsequence relation.

Proof:

Because the unique sorts of the above theorem form a complete lattice, the equivalence classes on the sorts must have a least upper bound. ■

3.3 Persistent Knowledge Store

The persistent knowledge store organizes and stores graph logic propositions in an efficient, vivid [EBBK89] architecture.

3.3.1 Knowledge Store Data Structures

The knowledge store data structures are implemented as objects in CLOS. They serve as the persistent knowledge store of WEB. It appears that either a non-persistent version or a version with secondary storage could be implemented in a manner which would be transparent with respect to its use in WEB. Thus, we only discuss the persistent version.

The data structures are:

- link — Links are the low level constructs, and they store both regular and inverted permission-link pairs against which they are connected.
- permission — Permissions store link pairs which they connect.
  - regular — The connected links are physically stored for regular permissions.
  - virtual — Virtual permissions are a function in WEB which calculates connected links.
- node — Nodes encapsulate the definitions of permissions and links.
- basic-link — Links, permissions, and nodes are all basic-links.
- pointer — A pointer is a link which also contains a "location", which is another link in the graph. The location can be set and changed and is used to parameterize the knowledge base.
- constructor — A constructor is a virtual permission generalized to a n-ary function which also can create constructs (links and permissions) in the WEB graph.

Chapter 4
Knowledge Base Programming

The percepts themselves may be shown to differ: but if each of us be asked to point out where his percept is, we point to an identical spot.
— *William James*

It is important that knowledge base programming be well defined. In knowledge intensive applications, the user must know under what conditions information can and cannot be obtained from the knowledge base. For this to occur, there must be a formal specification of how knowledge base programming works. We have chosen to use constructive type theory as a foundation for the theory of knowledge base programming because it:

1. has a strict type discipline,
2. is powerful enough for the task, yet can be made easy to use by implementing rule contruction algorithms,
3. can be used to separate type from structure information (as we have done) which leads to a cleaner notion of inheritance, and
4. can be implemented in a manner which leads to provable correct programs.

Because it is easier to write programs than proofs of of correctness, we use constructive type theory as the theory behind a functional programming language. This allows programs to be defined in the traditional manner, but it also yields a sequence of inference rules when the programs are executed. Because of the strong type discipline of SPIDER, this sequence of inference rules forms a proof of correctness. This could then be checked against an abstract specification by the programmer.[5] Even if compared manually against an abstract specification, this adds an extra level of certitude that the program is correct.

One way to access a knowledge base is to have a knowledge definition language and a knowledge manipulation language by analogy to the data definition language and data

[5] This can be done automatically, but we do not explore that here.

manipulation language of databases. However, persistent database programming languages have shown themselves to be more effective than separate definition and manipulation languages. We draw on results from database programming languages and define SPIDER as a knowledge base programming language. In addition, extensible databases have demonstrated that it is possible to develop databases other than from scratch. Because SPIDER is used both to develop applications (like a database programming language) and is extensible with new type constructors, data constructors, and internal access methods, SPIDER is best described as the first extensible knowledge base programming language.

We want a simple programming language. Rather than develop a large inclusive language for accessing the knowledge base, such as Machiavelli [BO90] or E [CDG+90], SPIDER was developed to only perform knowledge base related tasks and to be embedded in a larger functional language which would be used to create the other application programs. SPIDER is a restricted programming language (as Abiteboul proposes in the declarative paradigm [Abi89]) and has the constructs of a simple, strongly-typed, functional programming language [CW85, Jon87]. A knowledge base programming language must have certain *structural* and *behavioral* (functional) requirements in order to serve as an interface between knowledge-rich applications and a knowledge base. Structurally, it must contain association, taxonomic, and modularization constructs, and it must have a well-specified semantics. Behaviorally, it also must support both querying and reasoning.

This chapter contains the details of SPIDER. It shows how type inference rules can be used as a programming language (section 4.1), explains the type constructors which can be defined in SPIDER and gives algorithms to calculate their inference rules (section 4.2). We give an operational semantics for SPIDER in terms of constructive type theory inference rules (section 4.3) and introduce some of the advantages of using constructive type theory to describe inheritance (section 4.4).

4.1 Programming Using Inference Rules

Because constructive type theory is constructive mathematics, the inference rules can be used as the basis for a programming language. This is done for SPIDER, and here we show how the inference rules are used to evaluate a SPIDER program.

Looking in the context of the `tree-search` function (section 2.2.1), we can look at an application of the function *tree-search* where it looks for an element in the tree. We take as a specific example a binary tree of type BinTree(ThreeStooges) where ThreeStooges is a simple type of three elements. We apply the *tree-search* function:

$$\begin{aligned} \textit{tree-search} \equiv \lambda x.\lambda ele.\textit{BinTree-elim}(x, \lambda a.(ele \text{ EQUAL } a), \\ \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r)) \end{aligned}$$

to the binary tree node(leaf($larry$), leaf(moe)). The function is a lambda expression which has two parameters x and ele. It returns the expression obtained by evaluating the *BinTree-elim* function on the values passed to the parameters. *BinTree-elim* has three arguments: the main argument x, a lambda expression with one parameter, and a lambda expression with four parameters. The *BinTree-elim* function is defined by the computation rules (in psuedo-code) as:

```
BinTree-elim(x,leaf_abs,node_abs)
     if x is a leaf then
        apply leaf_abs as specified in the leaf-computation rule
     else if x is a node then
        apply node_abs as specified in the node-computation rule
     endif
```

where the proper data constructor (`leaf` or `node`) is determined by graph querying.

$$\begin{aligned} &\textit{tree-search}(\text{node}(\text{leaf}(larry), \text{leaf}(moe)), moe) \\ &\quad = (\lambda self.\lambda ele.\textit{BinTree-elim}(self, \lambda a.(ele \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r))) \\ &\qquad \text{node}(\text{leaf}(larry), \text{leaf}(moe)) \;\; moe \\ &\quad = \textit{BinTree-elim}(\textit{node}(\textit{leaf}(larry), \textit{leaf}(moe)), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r)) \end{aligned}$$

This gives us an elimination form which can be evaluated as specified by the computation rules. The *node-computation* rule tells that in $C[\text{node}(l, r)]$,

$$\begin{aligned} &\textit{BinTree-elim}(\text{node}(l, r), leaf\_abs, node\_abs) \\ &\qquad = \; node\_abs(l, r, \textit{BinTree-elim}(l, leaf\_abs, node\_abs), \textit{BinTree-elim}(r, leaf\_abs, node\_abs)) \end{aligned}$$

Thus, by filling in l, r, *leaf_abs* and *node_abs*, we have:

$$\begin{aligned}
&\mathit{BinTree\text{-}elim}(\mathit{node}(\mathit{leaf}(larry), \mathit{leaf}(moe)), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\quad = (\lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\qquad\quad \mathit{leaf}(larry) \quad \mathit{leaf}(moe)\\
&\qquad\quad \mathit{BinTree\text{-}elim}(\mathit{leaf}(larry), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\qquad\quad \mathit{BinTree\text{-}elim}(\mathit{leaf}(moe), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\quad \in C[\mathit{node}(\mathit{leaf}(larry), \mathit{leaf}(moe))]
\end{aligned}$$

Applying the lambda function (using β-reduction), we obtain:

$$\begin{aligned}
&\mathit{BinTree\text{-}elim}(\text{leaf}(larry), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\text{OR } \mathit{BinTree\text{-}elim}(\text{leaf}(moe), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))
\end{aligned}$$

still in type $C[\text{node}(\text{leaf}(larry), \text{leaf}(moe))]$. First, we try

$$\mathit{BinTree\text{-}elim}(\text{leaf}(larry), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r))$$

using the *leaf-computation* rule:

$$\begin{aligned}
&\mathit{BinTree\text{-}elim}(\text{leaf}(larry), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec_l.\lambda rec_r.(rec_l \text{ OR } rec_r))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = (\lambda a.(moe \text{ EQUAL } a))\ larry
\end{aligned}$$

which gives *false* by the definition of $EQUAL$ over the type. Now, we try

$$\mathit{BinTree\text{-}elim}(\text{leaf}(moe), \lambda a.(moe \text{ EQUAL } a), \lambda l.\lambda r.\lambda rec\_l.\lambda rec\_r.(rec\_l \text{ OR } rec\_r))$$

which yields $(\lambda a.(moe \text{ EQUAL } a))\ moe$ by the *leaf-computation* rule. This gives *true* by the definition of $EQUAL$. In our original form, we have *false* OR *true* which gives *true*. We now know, $true \in C[\mathit{leaf}(moe)]$, $true \in C[\mathit{node}(\mathit{leaf}(larry), \mathit{leaf}(moe))]$, $false \in C[\mathit{leaf}(larry)]$, and we have the result $\mathit{tree\text{-}search}(\text{node}(\text{leaf}(larry), \text{leaf}(moe)), moe) = true$.

We show how proofs can be extracted from the evaluation in section 4.3.

4.2 Rule Construction Algorithms

There are four kinds of type constructors allowed in SPIDER for knowledge base design — simple, recursive, inductive, and product — or any combination of them. Recursive type constructors have data constructors which are built from the type being defined, e.g., LIST(A), BINTREE(A), etc. Inductive type constructors have data constructors which are constructed inductively using a generalization of multi-valued attributes (explained in section 4.2.2). Simple type constructors are neither recursive or inductive. Product type constructors combine any two other type constructors in the manner described in section 4.2.3.

4.2.1 Recursive Types

The user defines a SPIDER type using `defstype`. The `defstype` form for the BinTree(A) example section 2.2.1 looks like:

```
defstype BinTree (A)
  leaf(A)                       = leaf
  node(BinTree(A),BinTree(A)) = treenode
```

where `leaf(A)` and `node(BinTree(A),BinTree(A))` describe the SPIDER data constructors for BinTree(A) in terms of the type variable A, and `leaf` and `treenode` give the name of the WEB program which builds the graph to be associated with each data constructor.

This creates the formation rule and two introduction rules for BinTree(A) given on p. 24. When each introduction rule is defined, a SPIDER function is created which will call the appropriate WEB program.

From these rules the elimination rule and computation rules are built as below, which is based on the process defined in [BCM88, Bac86b].

1. Find the recursive introduction variables in the introduction rules.

2. Create the elimination rule. We use the notation developed by Dijkstra [DF84] and used by Backhouse [Bac86a] for constructive type theory to make the description of the rule construction algorithm easier. Each conditional premise is denoted by the form $[\![\ premises \quad \triangleright \quad conclusion\ ]\!]$.

   BinTree-elimination

$$\frac{\begin{array}{ll} [\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] & \text{— type premise} \\ self \in BinTree(A) & \text{— major premise} \\ [\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]\ ]\!] & \text{— leaf premise} \\ [\![\ l \in BinTree(A) & \text{— node premise} \\ \quad r \in BinTree(A) & \\ \quad rec\_l \in C[l] & \\ \quad rec\_r \in C[r] & \\ \quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] & \\ ]\!] & \end{array}}{BinTree\text{-}elim(self, leaf\_abs, node\_abs) \ \in \ C[self]}$$

   The type premise and major premise are similar for all elimination rules we will consider. There is a minor premise for each introduction rule: the *leaf-premise* and the *node-premise*.

The elimination rule creates a form which is used to break apart elements of the types into their constituents. The manner in which it is done is specified by the arguments to the form. In this case, the form is *BinTree-elim*, and it is given an element of BINTREE as its first argument. The other arguments tell how to eliminate the top level data constructor (as specified by the computation rules). There is one parameter for each data constructor and thus for each introduction rule. The elimination form, here *BinTree-elim*, has four kinds of parameters:

(a) *self* — the first parameter is an element of the type *BinTree(A)*. Thus, it is either `leaf(?a)` or `node(?l,?r)`. This is the expression which is to be reduced to a canonical form as specified by the computation rules.

(b) VALue parameters — the value to be returned if the first argument is a constructor with no parameters. This does not occur in the BINTREE(A) type.

(c) Non-recursive abstractions — *leaf-abs* — a lambda abstraction that will be applied if *self* is bound to a constructor which has no recursive introduction variables. Each parameter in the abstraction is bound to an argument in the data constructor. The *leaf* data constructor fits this condition. Its argument is:

   (i) a — the variable to be bound to `?a`.

(d) Recursive abstractions — *node-abs* — a lambda abstraction which will be applied if *self* is bound to a constructor which has recursive introduction variables. First, the arguments of the data constructor are given as the first parameters of abstraction. Then, recurse variables are created for the abstraction which correspond to the recursive introduction variables. The *node* data constructor fits this condition. Its arguments are:

   (i) l — the variable to be bound to `?l`.

   (ii) r — the variable to be bound to `?r`.

   (iii) *rec-l* — This is created because `?l` is a recursive introduction variable. It defines how the type is recursed on via the `?l` argument of `node`.

   (iv) *rec-r* — This is created because `?r` is a recursive introduction variable. It defines how the type is recursed on via the `?r` argument of `node`.

3. Now, a computation rule is created for each introduction rule. There are two kinds of computation rules: one if a constructor does not have a recursive introduction variable (e.g., `leaf`) and one if it does (e.g., `node`).

(a) no recursive introduction variables — replace the major premise of the elimination rule with the premises of the introduction rule. For `leaf` this yields:

leaf-computation

$$
\begin{array}{l}
[\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\
a \in A \\
[\![\ a \in A \quad \triangleright \quad leaf\_abs(a)\ \in\ C[\mathrm{leaf}(a)]\ ]\!] \\
[\![\ l \in BinTree(A) \\
\quad r \in BinTree(A) \\
\quad rec\_l \in C[l] \\
\quad rec\_r \in C[r] \\
\quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r)\ \in\ C[\mathrm{node}(l, r)] \\
]\!] \\
\hline
\mathit{BinTree\text{-}elim}(\mathrm{leaf}(a), leaf\_abs, node\_abs) = leaf\_abs(a) \quad \in \quad C[\mathrm{leaf}(a)]
\end{array}
$$

(b) with recursive introduction variable — replace the major premise of the elimination rule with the premises of the introduction rule as was done above. The right hand side of the equality in the conclusion is a call to the recursive abstraction (*node-abs*) with each recursing parameter (*rec-l* and *rec-r*) bound to a (lazy) call of the elimination function where the first argument is the value bound to the associated recursive introduction variable. For `node`, this yields:

node-computation

$$
\begin{array}{l}
[\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\
l \in BinTree(A) \\
r \in BinTree(A) \\
[\![\ a \in A \quad \triangleright \quad leaf\_abs(a)\ \in\ C[\mathrm{leaf}(a)]\ ]\!] \\
[\![\ l \in BinTree(A) \\
\quad r \in BinTree(A) \\
\quad rec\_l \in C[l] \\
\quad rec\_r \in C[r] \\
\quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r)\ \in\ C[\mathrm{node}(l, r)] \\
]\!] \\
\hline
\mathit{BinTree\text{-}elim}(\mathrm{node}(l, r), leaf\_abs, node\_abs) \\
\qquad\qquad = node\_abs(l, r, \mathit{BinTree\text{-}elim}(l, leaf\_abs, node\_abs), \\
\qquad\qquad\qquad\qquad \mathit{BinTree\text{-}elim}(r, leaf\_abs, node\_abs)) \\
\qquad \in \ C[\mathrm{node}(l, r)]
\end{array}
$$

4. Then, a SPIDER function `BinTree-elim` is created which lazily evaluates expressions of type BinTree when given the appropriate WEB graph.

4.2.2 Inductive Types

We define inference rules for a type constructor which allows multi-valued attributes (and their generalizations) to be integrated into SPIDER. Instead of defining this as a SPIDER type, we modify the type inference rules to deal with set-valued variables and define the MVA(A) type constructor to refer to a subset of a type.

We then show how this can be used to define the SPIDER type SET and give a general algorithm for calculating the elimination and computation rules for an inductive type definition in SPIDER.

4.2.2.1 MVA Type

MVA(A) is a built in type constructor which allows disjunctive or conjunctive sets to be cleanly integrated into the type theory and alleviates set-value impedance mismatch [BM88] between the declarative knowledge base WEB and the functional programming language SPIDER. Because the WEB knowledge base allows for multi-valued attributes, it is necessary for SPIDER to handle sets of values. There are several ways to form the built-in type constructor, but we will just give one of them.[6]

MVA-formation

$$\frac{A\ type}{MVA(A)\ type}$$

This rule forms the MVA(A) type and states that if A is a type, then MVA(A) is a type.

$\emptyset$-introduction

$$\frac{}{\emptyset \in MVA(A)}$$

This introduces the empty value into MVA(A). This occurs if the specified attribute has no value at the node.

$\cup$-introduction

$$\frac{\begin{array}{l} a \in A \\ r \in MVA(A) \end{array}}{\{a\} \cup r\ \in MVA(A)}$$

[6] Technically, we should include congruence rules to state that the type MVA(A) is order-independent. However, we omit these for simplicity and because the implementation actually is order-dependent, though we do not wish to make use of the order-dependence when writing programs.

This introduces new values for an attribute. A single value for an attribute is represented as $\{a\} \cup \emptyset$.

MVA-elimination

$$
\dfrac{
\begin{array}{l}
[\![w \in \mathit{MVA(A)} \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\
x \in \mathit{MVA(A)} \\
b \in C[\emptyset] \\
[\![\; a \in A \\
\quad r \in \mathit{MVA(A)} \\
\quad i \in C[r] \\
\quad \triangleright \quad z(a, r, i) \; \in \; C[\{a\} \cup r] \\
]\!]
\end{array}
}{
\mathit{MVA\text{-}elim}(x, b, z) \; \in \; C[x]
}
$$

The elimination rule abstracts how to perform computations on the type. It contains expressions to be used by the computation rules to obtain a base value, b, and an induction step, z. The $\cup$-*abstraction*, z, must have three parameters. The third argument to z specifies the induction to be performed on the second argument. This is related to the recurse variables in traditional constructive type theory.

The computation rules are:

$$
\begin{aligned}
\mathit{MVA\text{-}elim}(\emptyset, b, z) \; &= \; b \;\; \in \;\; C[\emptyset] \\
\mathit{MVA\text{-}elim}(\{a\} \cup r, b, z) \; &= \; z(a, r, \mathit{MVA\text{-}elim}(r, b, z)) \;\; \in \;\; C[\{a\} \cup r]
\end{aligned}
$$

We can define the "$-$" function to remove an element from a MVA set of values.

$$
- \equiv \lambda s.\lambda x.\mathit{MVA\text{-}elim}(s, s, \lambda a.\lambda r.\lambda i.\ \mathit{if}\ \ (a\ \ \mathit{eq}\ \ x)\ \ r\ \ (\{a\} \cup r))
$$

Note that the value x does not have to be in the MVA set s for the function to work. If this is not what is wanted, the second argument to *MVA-elim* can be replaced by an error message.

Notation: We use **boldface** type for variables of the MVA type when it makes for clearer exposition, and we may write $\mathbf{x} \subseteq A$ for $\mathbf{x} \in MVA(A)$ in the inference rules.

4.2.2.2 Set

In order to define SET(A) in terms of MVA(A), we first need to define a single-valued attribute version of SET(A) called TAG.

TAG makes use of an type called IDS which is just a collection of distinct ids. These ids are a form of object identity as is found in object-oriented databases [KC86]. The entire purpose of IDS is to distinguish between the different tagged elements of TAG within constructive type theory. This will generalize from tags to sets and will allow SPIDER to store as distinct sets in the knowledge base those which might have the same members. This occurs because the knowledge base distinguishes between separate objects which may have the same mathematical expression, and we want the theory to follow. Basically, since we can store a set $\{a, b\}$ twice in the knowledge base, we need to distinguish between the two sets in SPIDER.

This touches on the issue of a type beinging *intensional* or *extensional* with regard to equality because SPIDER deals with sets using intensional equality (`eq` in Lisp) while mathematically sets are usually thought of extensionally (`equal` in Lisp). IDS can be defined using constructive type theory, but we will not do it here.

Tag-formation

$$\frac{A\ type}{Tag(A)\ type}$$

The TAG formation rule states: if A is a type, then *Tag(A)* is a type.

tag-introduction

$$\frac{\begin{array}{l} a \in A \\ n \in Ids \end{array}}{\mathrm{tag}(a, n) \ \in \ Tag(A)}$$

If a is a member of the type A, and n is a member of *Ids*, then **tag**(a, n) is a member of the type *Tag(A)*.

The conclusion of the elimination rule is:

$$\textit{Tag-elim}(x, \textit{tag\_abs}) \ \in \ C[x].$$

Tag-elim is a "form" of two arguments, and it is in the class of objects generated by the first argument. The second argument specifies how to calculate a certain value when given x. How the argument *tag_abs* is used is defined by the computation rules. The elimination form is evaluated using lazy, normal-order reduction. The complete elimination inference rule is:

Tag-elimination

$$\begin{array}{l}
[\![w \in Tag(A) \quad \triangleright \quad C[w]\ type]\!] \text{— type premise} \\
x \in Tag(A) \qquad\qquad\qquad\qquad \text{— major premise} \\
[\![\ a \in A \qquad\qquad\qquad\qquad\quad \text{— tag premise} \\
\quad n \in Ids \\
\quad \triangleright \quad tag_abs(a, n) \ \in \ C[\mathrm{tag}(a, n)] \\
]\!] \\
\hline
Tag\text{-}elim(x, tag_abs) \ \in \ C[x]
\end{array}$$

The *type premise* defines the class generated by the type (indexed by objects in the type), and the *major premise* specifies an arbitrary element of the type to be reasoned with. The *tag_abs* term will be defined by the individual programs on the type, but they must be of the type specified in its premise, the *tag premise*:

$$\begin{array}{l}
[\![\ a \in A \\
\quad n \in Ids \\
\quad \triangleright \quad tag_abs(a, n) \ \in \ C[\mathrm{tag}(a, n)] \\
]\!]
\end{array}$$

which states the form $tag\_abs(a, n)$ is in the class of results generated by $tag(a, n)$.

Since Tag has only one data constructor, there is one computation rule. It is of the form

$$Tag\text{-}elim(< constructor >, tag\_abstraction) = < value >$$

which holds in the class of canonical expressions generated by the constructor. Thus the full conclusion is

$$Tag\text{-}elim(< constructor >, tag\_abstraction) = < value > \ \in \ C[< constructor >].$$

Because, the premises of a computation rule are very easy to calculate from the elimination rule, we will omit the premises of the computation rule and just give its conclusions. The computation rule **tag-computation** is:

$$Tag\text{-}elim(\mathrm{tag}(a, n), tag\_abstraction) \ = \ tag\_abstraction(a, n) \ \in \ C[\mathrm{tag}(a, n)]$$

All of these rules are defined or calculated in SPIDER by the `defstype` form:

```
defstype Tag (A)
   tag(A,Ids)  = wTag
```

where `wTag` is a graph constructor in WEB with two parameters (the same number as `tag`).

TAG(A) is not a particularly interesting type, so we will only give one simple example function for it in SPIDER. The function `tag-value` returns the single value of a tag.

```
defsfun tag-value Tag(A)
    ()
  tag(?a,?n) => ?a
```

This is equivalent to the lambda expression

$$\lambda x.\mathit{Tag\text{-}elim}(x, \lambda a.\lambda n.a)$$

We now create the type SET1 which is identical to TAG except that we replace the data constructor `tag` with `ele` and change membership of its first argument, a, from A to MVA(A).

Set1-formation

$$\frac{A\ \mathit{type}}{\mathit{Set}1(A)\ \mathit{type}}$$

ele-introduction

$$\frac{\begin{array}{l}\mathbf{a} \subseteq A \\ n \in \mathit{Ids}\end{array}}{\mathrm{ele}(\mathbf{a}, n)\ \in\ \mathit{Set}1(A)}$$

The variable **a** is an element of MVA(A).

Set1-elimination

$$\frac{\begin{array}{l}[\![w \in \mathit{Set}1(A) \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\ x \in \mathit{Set}1(A) \\ [\![\ \mathbf{a} \subseteq A \\ \quad n \in \mathit{Ids} \\ \quad \triangleright \quad z(\mathbf{a}, n)\ \in\ C[\mathrm{ele}(\mathbf{a}, n)] \\ ]\!]\end{array}}{\mathit{Set1\text{-}elim}(x, z)\ \in\ C[x]}$$

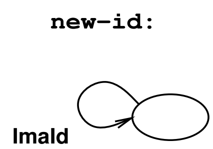
The computation rule **ele-computation** is:

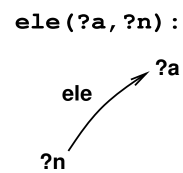
$$\mathit{Set1\text{-}elim}(\mathrm{ele}(\mathbf{a}, n), z)\ =\ z(\mathbf{a}, n)\ \in\ C[\mathrm{ele}(\mathbf{a}, n)]$$

Note that we did not have to create a data constructor for the empty set in SET1. We instead consider the empty set of SET1 to occur when $\emptyset$ from MVA(A) is the first argument to `ele`, i.e., `ele(∅,?n)`.

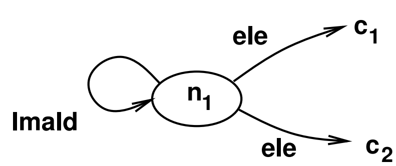
Now, go back to consider what `ele(∅,?n)` means with respect to the knowledge base. First, consider how sets of type SET1 are formed.

1. Create a new id n_1, say by calling a function `new-id`.
2. Create a SET1 set with members c_1 and c_2, where c_1, c_2 are constants. Do this by calling `ele` twice:
```
>> ele(c1,n1)
>> ele(c2,n1)
```
3. This will create two entries in the knowledge base where c_1 and c_2 are multi-valued attributes off of n_1. For concreteness, consider the functions `new-id` and `ele` to create the following graphs:





Thus our knowledge base would contain:



In this case, the expression `ele(∅,?n)` refers to zero applications of `ele` to the id `?n`, which is the value returned by `new-id`. Thus `new-id` is the empty set of SET1.

More generally,

Theorem Let θ be a data constructor of type Θ and arity p which:

1. has arguments $m_1, m_2, \ldots, m_{p-1}$ which are each of type $MVA(\Theta_i)$ with Θ_i a type, for $1 \leq i \leq p$,
2. θ also has an argument n of type Θ', without loss of generality say the p^{th} argument, and
3. Θ' is *not* of type MVA (which can be made precise),

then if c' is a value in Θ', then $\theta(\emptyset, \emptyset, \ldots, \emptyset, c') \ = \ c'$.

Proof:

The expression $\theta(\emptyset, \emptyset, \ldots, \emptyset, c')$ can only hold if θ were called zero times with c' as the last argument. This is the same as only having c' in the knowledge base. ∎

Notice that `new-id` returns a value of type IDS and `ele(∅,new-id())` returns a value of type SET1, and those values are identical in the knowledge base. In most strongly typed languages, this would cause problems. However, the knowledge base in WEB does not store type information, and thus this is no different than any other retrieval. In fact, it is an advantage because data can be stored as one type and retrieved as another. Instead of forcing SPIDER to choose between the types IDS and SET1 for a value of `new-id`, SPIDER is informed that the `new-id` values from IDS are included in the type.[7] The desired type can be sufficiently determined from the context in which it is used (see section 4.4.1).

Now consider an application of SET1 by defining the member function over sets:

$$member_1 \equiv \lambda s.\lambda x.Set1\text{-}elim(s, \lambda \mathbf{a}.\lambda t.MVA\text{-}elim(\mathbf{a}, false, \lambda a.\lambda r.\lambda i.\ if\ \ (a\ \ eq\ \ x)\ \ true\ \ \ i))$$

To define this in SPIDER using `defsfun` we need to call `MVA-elim` directly.

```
defsfun member1 Set1(A)
        (?x)
   ele(?a, ?n) => MVA-elim (?a,
                    empty() => false(),
                    mva-union(?a, ?r) => if ?a eq ?x
                                            true()
                                          recurse (?r))
```

However, this syntax can get tiresome, so SPIDER allows for definition as:

```
defsfun member1 Set1(A)
        (?x)
   ele(empty(), ?n) => false()
   ele(?a,?n)::?next
    => if ?a eq ?n
         true()
       recurse(?next)
```

where `empty()` refers to $\emptyset$. But, because `ele(empty(),?n)` has the same value as `new-id`, and SPIDER was told that elements of IDS created by `new-id` are included in SET1, we can substitute the polymorphic constructor `new-id` for `ele(empty(),?n)`. To improve readability, SPIDER also allows for aliasing of polymorphic constructors, thus we can alias `new-id` as `empty-set`. This results in the final definition for `member1`:

```
defsfun member1 Set1(A)
        (?x)
   empty-set() => false()
   ele(?a,?n)::?next
    => if ?a eq ?n
         true()
```

[7] Actually because IDS only has one data constructor all elements of IDS are included in SET1. If IDS had another data constructor which was not included in SET1, then only the `new-id` constructor would be included. This is done using a *data constructor subsumption* rule as explained in section 4.4.

```
recurse(?next)
```

Now having considered the ramifications adding the MVA type constructor had on the knowledge base, we must consider the effects on SPIDER of noticing that `new-id` and `ele(∅,new-id())` are identical except for type information, and this information is given to SPIDER.[8] SPIDER must be told that *new-id()* $=$ *ele*$(\emptyset,$ *new-id()*$)$. This occurs in the type SET1, and thus the full congruence rule is:

ele-base-equality

$$\frac{\text{new-id}() = n \ \in \ Ids}{\text{ele}(\emptyset, \text{new-id}()) = n \ \in \ Set1}$$

This results in an elimination rule with an added premise for `new-id`: $b \in C[\text{new-id}()]$ and with premises for the **Ids-subsumption** and **ele-base-equality** rules.

Set1-elimination

$$\frac{\begin{array}{l} [\![w \in Set1(A) \quad \triangleright \quad C[w] \ type]\!] \\ x \in Set1(A) \\ [\![n \in Ids \quad \triangleright \quad n \in Set1]\!] \\ [\![\text{new-id}() = n \ \in \ Ids \quad \triangleright \quad \text{ele}(\emptyset, \text{new-id}()) = n \ \in \ Set1]\!] \\ b \in C[\text{new-id}()] \\ [\![\ \mathbf{a} \subseteq A \\ \quad n \in Ids \\ \quad \triangleright \quad z(\mathbf{a}, n) \ \in \ C[\text{ele}(\mathbf{a}, n)] \\ ]\!] \end{array}}{\textit{Set1-elim}(x, b, z) \ \in \ C[x]}$$

However, because $\emptyset$ can occur not only in $ele(\emptyset, n)$ as the initial argument, x, to *Set1-elim*, but also as the end value of a nonempty MVA union, we want to insure that the *MVA-elim* form bound to z has the same base value b as *Set1-elim*.

To do this, the *MVA-elim* form is included in the *Set1-elim* rule. This corresponds to the syntactic conversion above where the `MVA-elim` form was included in the `defsfun` for `member1`. This gives an elimination rule of:

[8] This could be done automatically, but in the current implementation it is done by the user when defining SET1.

Set1-elimination

$$\begin{array}{l}
[\![w \in Set1(A) \quad \triangleright \quad C[w]\ type]\!] \\
x \in Set1(A) \\
[\![n \in Ids \quad \triangleright \quad n \in Set1]\!] \\
[\![\text{new-id}() = n \ \in \ Ids \quad \triangleright \quad \text{ele}(\emptyset, \text{new-id}()) = n \ \in \ Set1]\!] \\
b \in C[\text{new-id}()] \\
[\![\ a \in A \\
\quad \mathbf{r} \subseteq A \\
\quad i \in C[\mathbf{r}] \\
\quad n \in Ids \\
\quad \triangleright \quad z(a, \mathbf{r}, i, n) \ \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)] \\
]\!] \\
\hline
\textit{Set1-elim}(x, b, z) \ \in \ C[x]
\end{array}$$

The new computation rules are:

$$\textit{Set1-elim}(\text{new-id}(), b, z) \ = \ b \ \in \ C[\text{new-id}()]$$

$$\textit{Set1-elim}(\text{ele}(\{a\} \cup \mathbf{r}, n), b, z) \ = \ z(a, \mathbf{r}, \textit{Set1-elim}(\text{ele}(\mathbf{r}, n), b, z), n) \ \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)]$$

This completes the definition of SET1. This **Set1-elimination** rule and computation rules are created automatically by SPIDER, and are used to define access methods on the type.

4.2.2.3 Inductive Rule Algorithm

Before presenting the algorithm for building inductive inference rules, we need to decide how to handle data constructors with multiple arguments of type MVA(A). The two choices are *dependent*, where the values are grouped together (e.g., in FEATURESTRUCTURE(A) in section 6.2), and *independent* where the values are can vary separately (e.g., some type using SET(A)×SET(A)). There was not this decision to be made for recursive types because treating them independently would have lead to information loss.[9]

This is why `recurse` takes two arguments only when they come from separate (recursive) data constructors. This does not occur with inductive types because all non-MVA arguments are constant for one instance of a data constructor (by definition). The problem occurs with inductive types when treating arguments independently. This will only work when the multi-valued attributes are part of the disjoint graphs in the knowledge base. Rather than checking for this situation and allowing independent recursion when it occurs,

[9] Consider a binary tree with information in the nodes, say `node(?left,?x,?right)`, with type description $node : BT(A) \times A \times BT(A)$. If the first and third arguments were recursed simultaneously, i.e., `(recurse ?left ?right)`, only one of the subnodes' datum, `?x`, could be retained to be used by the (recursive) calling function.

SPIDER only allows for dependent recursion on inductive types.[10] This is why `recurse` takes data constructor arguments for recursive types and takes induction variables (`?next` in above examples) for inductive ones.[11]

The induction rule algorithm is:

1. Find the introduction rules which use multivalued attributes, and keep track of their base cases.

2. Create the elimination rule.

Set1-elimination

$$
\begin{array}{ll}
[\![w \in Set1(A) \quad \triangleright \quad C[w]\ type]\!] & \text{— type premise} \\
x \in Set1(A) & \text{— major premise} \\
[\![\textit{new-id()} = n \ \in \ \textit{Ids} \quad \triangleright \quad \textit{ele}(\emptyset, \textit{empty-set()}) = n \ \in \ \textit{Set1}]\!] & \text{— ele-base-equality} \\
\textit{empty-set-val} \in C[\text{new-id}()] & \text{— empty-set premise} \\
[\![\ a \in A & \text{— ele premise} \\
\quad \mathbf{r} \subseteq A & \\
\quad i \in C[\mathbf{r}] & \\
\quad n \in Ids & \\
\quad \triangleright \quad \textit{ele-abs}(a, \mathbf{r}, i, n) \ \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)] & \\
]\!] & \\
\hline
\textit{Set1-elim}(x, \textit{empty-set-val}, \textit{ele-abs}) \ \in \ C[x] &
\end{array}
$$

The type premise and major premise are the same as the recursive case. There is a minor premise for each introduction rule: the *empty-set-premise* and the *ele-premise*. There is also an *ele-base-equality* premise which corresponds to the *ele-base-equality* congruence rule.

The elimination rule creates a form which is used to break apart elements of the types into their constituents. The manner in which it is done is specified by the arguments to the form. In this case, the form is *Set1-elim*, and it is given an element of SET1 as its first argument. The other arguments tell how to eliminate the top level data constructor (as specified by the computation rules). There is one parameter for each data constructor and thus for each introduction rule. The elimination form has four kinds of parameters:

(a) *self* — the first parameter is an element of the type *Set1(A)*. Thus, it is either `empty-set()` or `ele(?a,?n)`. This is the expression which is to be reduced to a canonical form as specified by the computation rules.

---

[10] This does not cause any practical problem because disjoint graphs can always be created by using separate data constructors.

[11] The same form `recurse` is used for inductive types rather than, say `iterate`, because `recurse` actually linearizes the MVA set and performs induction on its length, thus effectively recursing over the induction variable. Because only induction types have iteration variables, no confusion should result.

(b) VALue parameters — empty-set-val — the value to be returned if the constructor has no parameters. In this case, *empty-set-val* is also a base value.

(c) Non-inductive abstractions — a lambda abstraction that will be applied if *self* is bound to a constructor which has no inductive introduction variables. Each parameter in the abstraction is bound to an argument in the data constructor. This does not occur in the SET1(A) type.

(d) Inductive abstractions — *ele-abs* — a lambda abstraction which will be applied if *self* is bound to a constructor which has introduction variables of type MVA. First, the arguments of the data constructor are given as parameters of abstraction. Then, inductive variables are created for the abstraction which correspond to the MVA introduction variables. The *ele* data constructor fits this condition. The abstraction arguments are:

 (i) a — an element in the type A.

 (ii) $\mathbf{r}$ — the remaining elements in *?a* (not including a).

 (iii) i — This is created because `?a` is a inductive introduction variable. It defines how induction is performed on the type.

 (iv) n — This is an element of IDS.

3. Now, a computation rule is created for each introduction rule. There are two kinds of computation rules: one if a constructor does not have an induction introduction variable (e.g., `empty-set`) and one if it does (e.g., `ele`).

 (a) no induction introduction variables — replace the major premise of the elimination rule with the premises of the introduction rule. For `empty-set` this yields:

empty-set-computation

$$\dfrac{\begin{array}{l} [\![w \in Set1(A) \quad \triangleright \quad C[w]\ type]\!] \\ [\![\textit{new-id()} = n \ \in \ \textit{Ids} \quad \triangleright \quad \textit{ele}(\emptyset, \textit{empty-set()}) = n \ \in \ \textit{Set1}]\!] \\ \textit{empty-set-val} \in C[\text{new-id}()] \\ [\![\ a \in A \\ \quad \mathbf{r} \subseteq A \\ \quad i \in C[\mathbf{r}] \\ \quad n \in Ids \\ \quad \triangleright \quad \textit{ele-abs}(a, \mathbf{r}, i, n) \ \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)] \\ ]\!] \end{array}}{\textit{Set1-elim}(\text{new-id}(), \textit{empty-set-val}, \textit{ele-abs}) \ = \ \textit{empty-set-val} \ \in \ C[\text{new-id}()]}$$

 (b) with induction introduction variable — replace the major premise of the elimination rule with the premises of the introduction rule as was done above. The right hand side of the equality in the conclusion is a call to the induction abstraction (*ele-abs*) with the induction parameter i bound to a (lazy) call of the elimination function where the first argument is the value bound to a construction of the data constructor on the remaining elements of MVA form. For `ele`, this yields:

ele-computation

$$
\begin{array}{l}
[\![w \in Set1(A) \quad \triangleright \quad C[w]\ type]\!] \\
a \in A \\
\mathbf{r} \subseteq A \\
i \in C[\mathbf{r}] \\
n \in Ids [\![\textit{new-id()} = n \ \in \ \textit{Ids} \quad \triangleright \quad \textit{ele}(\emptyset, \textit{empty-set()}) = n \ \in \ \textit{Set1}]\!] \\
\textit{empty-set-val} \in C[\text{new-id}()] \\
[\![\ a \in A \\
\quad \mathbf{r} \subseteq A \\
\quad i \in C[\mathbf{r}] \\
\quad n \in Ids \\
\quad \triangleright \quad \textit{ele-abs}(a, \mathbf{r}, i, n) \ \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)] \\
]\!] \\
\hline
\textit{Set1-elim}(\text{ele}(\{a\} \cup \mathbf{r}, n), \textit{empty-set-val}, \textit{ele-abs}) \\
\qquad\qquad\qquad = \ \textit{ele-abs}(a, \mathbf{r}, \textit{Set1-elim}(\text{ele}(\mathbf{r}, n), \textit{empty-set-val}, \textit{ele-abs}), n) \\
\qquad\qquad \in \ C[\text{ele}(\{a\} \cup \mathbf{r}, n)]
\end{array}
$$

4. Then, a SPIDER function `Set1-elim` is created which lazily evaluates expressions of type SET1 when given the appropriate WEB graph.

4.2.3 Product Types

So far we have considered only functions which depend upon the type of the first argument. In this section we generalize to functions which depend upon the type of their first two arguments.

This is more complex than might first be imagined because the types may be recursive. The first (insufficient) approach is to take the cartesian product of two types and write SPIDER programs on the product type.

For example, consider the definition of the function "and" for BOOLEAN × BOOLEAN:

```
defsfun and <Boolean,Boolean>
        ()
  true() true() => true()
  true() false() => false()
  false() true() => false()
  false() false() => false()
```

This is operationally equivalent [Jon87] to the function:

```
defsfun and1 Boolean
        (?x)
  true() => case ?x
```

```
                true() => true()
                false() => false()
     false() => case ?x
                 true() => false()
                 false() => false()
```

which is translated to

$$and1 \equiv \lambda self.\lambda x.\mathit{Boolean\text{-}elim}(self, (\lambda x.\mathit{Boolean\text{-}elim}(x, true, false))\,(x),\\ (\lambda x.\mathit{Boolean\text{-}elim}(x, false, false))\,(x))$$

which is what we want. However, this does not work in general for recursive or inductive types.

4.2.3.1 Recursive Product Types

Consider the function `list-equal`:

```
defsfun list-equal <List(A),List(A)>
        ()
   null() null() => true()
   null() cons(_, _) => false()
   cons(_,_) null() => false()
   cons(?a1,?l1) cons(?a2,?l2) =>
       if ?a1 eq ?a2  true()  recurse(?l1,?l2)
```

Here we wish to recurse both lists simultaneously and compare them element by element. This is not possible if we just take the cartesian product of the two types. We must modify the type to allow for simultaneous recursions. Instead of a cartesian product, we create a new type constructor PRODUCT(LIST(A),LIST(A)) and calculate its formation rule, introduction rules, elimination rule, and computation rules.

The resulting type PRODUCT(LIST(A),LIST(A)) is:

Product(List(A),List(A))-formation

$$\frac{\mathit{List}(A)\ \mathit{type}}{\mathit{Product}(\mathit{List}(A), \mathit{List}(A))\ \mathit{type}}$$

This defines the PRODUCT(A,B) type for LIST(A). The four introduction rules in PRODUCT(LIST(A),LIST(A)) are created by taking the cartesian product of the sets of assumptions for each introduction rule in LIST(A) for the assumptions of the new rules. The conclusion states that the pair of constructors is in PRODUCT(LIST(A),LIST(A)).

nil-nil-introduction

$$\frac{}{\langle \text{nil}, \text{nil} \rangle \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A))}$$

nil-cons-introduction

$$\frac{\begin{array}{l} a \in A \\ l \in \mathit{List}(A) \end{array}}{\langle \text{nil}, \text{cons}(a, l) \rangle \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A))}$$

cons-nil-introduction

$$\frac{\begin{array}{l} a \in A \\ l \in \mathit{List}(A) \end{array}}{\langle \text{cons}(a, l), \text{nil} \rangle \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A))}$$

cons-cons-introduction

$$\frac{\begin{array}{l} a_1 \in A \\ l_1 \in \mathit{List}(A) \\ a_2 \in A \\ l_2 \in \mathit{List}(A) \end{array}}{\langle \text{cons}(a_1, l_1), \text{cons}(a_2, l_2) \rangle \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A))}$$

The elimination rule is created using the algorithm of section 4.2 except for the last premise of the rule. This premise, corresponding to the **cons-cons-introduction** rule, is different because `cons` contains two recursive introduction variables, i.e., $l_1, l_2 \in \mathit{List}(A)$.

When in the pair $\langle \theta_1, \theta_2 \rangle$, both data constructors θ_1 and θ_2 are recursive, we want the corresponding abstraction to allow recursion on either θ_1 or θ_2 or on both of them simultaneously. This results in the elimination rule:

Product(List(A),List(A))-elimination

$$
\dfrac{
\begin{array}{ll}
[\![w \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A)) \quad \triangleright \quad C[w]\ \mathit{type}]\!] & \text{— type premise} \\
x \in \mathit{Product}(\mathit{List}(A), \mathit{List}(A)) & \text{— major premise} \\
\mathit{nil\_val} \in C[\langle \mathrm{nil}, \mathrm{nil} \rangle] & \text{— nil-nil-premise} \\
[\![\ a \in A & \text{— nil-cons-premise} \\
\quad l \in \mathit{List}(A) & \\
\quad \mathit{rec\_l} \in C[\langle \mathrm{nil}, l \rangle] & \\
\quad \triangleright \quad \mathit{nil\text{-}cons\_abs}(a, l, \mathit{rec\_l}) \ \in \ C[\langle \mathrm{nil}, \mathrm{cons}(a, l) \rangle] & \\
]\!] & \\
[\![\ a \in A & \text{— cons-nil-premise} \\
\quad l \in \mathit{List}(A) & \\
\quad \mathit{rec\_l} \in C[\langle l, \mathrm{nil} \rangle] & \\
\quad \triangleright \quad \mathit{cons\text{-}nil\_abs}(a, l, \mathit{rec\_l}) \ \in \ C[\langle \mathrm{cons}(a, l), \mathrm{nil} \rangle] & \\
]\!] & \\
[\![\ a_1 \in A & \text{— cons-cons-premise} \\
\quad l_1 \in \mathit{List}(A) & \\
\quad a_2 \in A & \\
\quad l_2 \in \mathit{List}(A) & \\
\quad \mathit{rec\_l}_1 \in C[\langle l_1, \mathrm{cons}(a_2, l_2) \rangle] & \\
\quad \mathit{rec\_l}_2 \in C[\langle \mathrm{cons}(a_1, l_1), l_2 \rangle] & \\
\quad \mathit{rec\_l}_3 \in C[\langle l_1, l_2 \rangle] & \\
\quad \triangleright \quad \mathit{cons\text{-}cons\_abs}(a_1, l_1, a_2, l_2, \mathit{rec\_l}_1, \mathit{rec\_l}_2, \mathit{rec\_l}_3) & \\
\qquad\qquad \in \ C[\langle \mathrm{cons}(a_1, l_1), \mathrm{cons}(a_2, l_2) \rangle] & \\
]\!] &
\end{array}
}{
\begin{array}{c}
\mathit{Product(List(A),List(A))\text{-}elim}(x, \mathit{nil\_val}, \mathit{nil\text{-}cons\_abs}, \mathit{cons\text{-}nil\_abs}, \\
\mathit{cons\text{-}cons\_abs}) \ \in \ C[x]
\end{array}
}
$$

In the *cons-cons-premise* we created a third *recurse* variable $\mathit{rec\_l}_3$ to correspond to the case when we recurse down both lists simultaneously. The four computation rules are shown in Figure 4.

4.2.3.2 Type Product Algorithm for Recursive Types

The product types are not explicitly defined by the user, but are created automatically when a function is defined as a SPIDER method with multiple typed arguments. The type constructor is stored to ensure that the type inference rule calculation only occurs once. The algorithm which computes the type product inference rules is implemented by passing the Algorithm for Recursive Types (section 4.2.1) a collection of introduction rules created as the cross product of the original introduction rules, with appropriate renaming of variables. The new type constructor is automatically stored in SPIDER as is shown in section 4.3.3.

When a type product is created the Algorithm for Recursive Types must also create a recurse variable for simultaneous recursions when the data constructors being crossed are both recursive. This occurs in step (2d) of the algorithm on p. 55. In addition, its

$$Product(List(A),List(A))\text{-}elim(\langle nil, nil\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs) = nil\_val \in C[\langle nil, nil\rangle]$$

$$\begin{aligned}&Product(List(A),List(A))\text{-}elim(\langle nil, cons(a,l)\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs)\\ &\quad = nil\text{-}cons\_abs(a, l, Product(List(A),List(A))\text{-}elim(\langle nil, l\rangle)) \in C[\langle nil, cons(a,l)\rangle]\end{aligned}$$

$$\begin{aligned}&Product(List(A),List(A))\text{-}elim(\langle cons(a,l), nil\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs)\\ &\quad = cons\text{-}nil\_abs(a, l, Product(List(A),List(A))\text{-}elim(\langle l, nil\rangle)) \in C[\langle cons(a,l), nil\rangle]\end{aligned}$$

$$\begin{aligned}&Product(List(A),List(A))\text{-}elim(\langle cons(a_1,l_1), cons(a_2,l_2)\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs)\\ &= cons\text{-}cons\_abs(a_1, l_1, a_2, l_2,\\ &\qquad\qquad Product(List(A),List(A))\text{-}elim(\langle l_1, cons(a_2,l_2)\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs),\\ &\qquad\qquad Product(List(A),List(A))\text{-}elim(\langle cons(a_1,l_1), l_2\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs),\\ &\qquad\qquad Product(List(A),List(A))\text{-}elim(\langle l_1, l_2\rangle, nil\_val, nil\text{-}cons\_abs, cons\text{-}nil\_abs, cons\text{-}cons\_abs)\\ &\qquad\qquad)\\ &\in C[\langle cons(a_1,l_1), cons(a_2,l_2)\rangle]\end{aligned}$$

Figure 4: List Product Computation Rules

computation rule must define the the right hand side argument to call the elimination form on the pair of recursive introduction variables which are used to specify that simultaneous recursion. If the recursive data constructor has more than one recursive introduction variable, this must be done for each variable, when the constructor is crossed with another recursive data constructor.

4.2.3.3 Inductive Product Types

Now, consider the product of two inductive types.

Product(Set1(A),Set1(A))-formation

$$\frac{Set1(A)\ type}{Product(Set1(A), Set1(A))\ type}$$

emptyset-emptyset-introduction

$$\frac{}{\langle \mathrm{emptyset}(), \mathrm{emptyset}()\rangle \in Product(Set1(A), Set1(A))}$$

We use *emptyset* as an alias for *new-id*.

emptyset-ele-introduction

$$\frac{\begin{array}{l}\mathbf{a} \subseteq A \\ n \in Ids\end{array}}{\langle \mathrm{emptyset}(), \mathrm{ele}(\mathbf{a}, n)\rangle \in Product(Set1(A), Set1(A))}$$

ele-emptyset-introduction

$$\frac{\begin{array}{l}\mathbf{a} \subseteq A \\ n \in Ids\end{array}}{\langle \mathrm{ele}(\mathbf{a}, n), \mathrm{emptyset}()\rangle \in Product(Set1(A), Set1(A))}$$

ele-ele-introduction

$$\frac{\begin{array}{l}\mathbf{a_1} \subseteq A \\ t_1 \in Ids \\ \mathbf{a_2} \subseteq A \\ t_2 \in Ids\end{array}}{\langle \mathrm{ele}(\mathbf{a_1}, t_1), \mathrm{ele}(\mathbf{a_2}, t_2)\rangle \in Product(Set1(A), Set1(A))}$$

Product(Set1(A),Set1(A))-elimination

$$
\begin{array}{ll}
[\![w \in \mathit{Product}(\mathit{Set1}(A), \mathit{Set1}(A)) \quad \triangleright \quad C[w] \ \mathit{type}]\!] & \text{— type premise} \\
x \in \mathit{Product}(\mathit{Set1}(A), \mathit{Set1}(A)) & \text{— major premise} \\
\mathit{emptyset\_val} \in C[\langle \text{emptyset}(), \text{emptyset}() \rangle] & \text{— emptyset-emptyset-premise} \\
[\![\ a \in A & \text{— emptyset-ele-premise} \\
\quad \mathbf{r} \subseteq A & \\
\quad t \in \mathit{Ids} & \\
\quad i \in C[\langle \text{emptyset}(), \text{ele}(\mathbf{r}, n) \rangle] & \\
\quad \triangleright \quad \mathit{emptyset\text{-}ele\_abs}(a, \mathbf{r}, t, i) \ \in \ C[\langle \text{emptyset}(), \text{ele}(\{a\} \cup \mathbf{r}, n) \rangle] & \\
]\!] & \\
[\![\ a \in A & \text{— ele-emptyset-premise} \\
\quad \mathbf{r} \subseteq A & \\
\quad t \in \mathit{Ids} & \\
\quad i \in C[\langle \text{ele}(\mathbf{r}, n), \text{emptyset}() \rangle] & \\
\quad \triangleright \quad \mathit{ele\text{-}emptyset\_abs}(a, \mathbf{r}, t, i) \ \in \ C[\langle \text{ele}(\{a\} \cup \mathbf{r}, n), \text{emptyset}() \rangle] & \\
]\!] & \\
[\![\ a_1 \in A & \text{— ele-ele-premise} \\
\quad \mathbf{r_1} \subseteq A & \\
\quad t_1 \in \mathit{Ids} & \\
\quad a_2 \in A & \\
\quad \mathbf{r_2} \subseteq A & \\
\quad t_2 \in \mathit{Ids} & \\
\quad i_1 \in C[\langle \text{ele}(\mathbf{r_1}, n), \text{ele}(\{a_2\} \cup \mathbf{r_2}, t_2) \rangle] & \\
\quad i_2 \in C[\langle \text{ele}(\{a_1\} \cup \mathbf{r_1}, t_1), \text{ele}(\mathbf{r_2}, t_2) \rangle] & \\
\quad i_3 \in C[\langle \text{ele}(\mathbf{r_1}, n), \text{ele}(\mathbf{r_2}, t_2) \rangle] & \\
\quad \triangleright \quad \mathit{ele\text{-}ele\_abs}(a_1, \mathbf{r_1}, t_1, a_2, \mathbf{r_2}, t_2, i_1, i_2, i_3) & \\
\qquad\qquad \in \ C[\langle \text{ele}(\{a_1\} \cup \mathbf{r_1}, t_1), \text{ele}(\{a_2\} \cup \mathbf{r_2}, t_2) \rangle] & \\
]\!] & \\
\hline
\mathit{Product(Set1(A),Set1(A))\text{-}elim}(x, \mathit{emptyset\_val}, \mathit{emptyset\text{-}ele\_abs}, \mathit{ele\text{-}emptyset\_abs}, & \\
\qquad \mathit{ele\text{-}ele\_abs}) \ \in \ C[x] &
\end{array}
$$

In the *ele-ele-premise* we created a third *induction* variable i_3 to correspond to the case when we iterate down both sets simultaneously. SPIDER allows each data constructor to be iterated over independently because different data constructor instances must have distinct graphs.[12] We also reorder some of the abstraction arguments, placing the induction variables at the end, to simplify part of the product algorithm. The four computation rules are shown in Figure 5.

---

[12] Technically, there would be only one graph in the knowledge base if the instances are the same value generated by the same data constructor, but they are retrieved separately from the knowledge base by SPIDER in the implementation of the PRODUCT(A,A) type, so there is still no problem.

$$\textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{emptyset()}, \textit{emptyset()}\rangle, \textit{emptyset\_val}, \textit{emptyset-ele\_abs}, \textit{ele-emptyset\_abs}, \textit{ele-ele\_abs})$$
$$= \textit{emptyset\_val} \in C[\langle \textit{emptyset()}, \textit{emptyset()}\rangle]$$

$$\textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{emptyset()}, \textit{ele}(\{a\}\cup\mathbf{r}, n)\rangle, \textit{emptyset\_val}, \textit{emptyset-ele\_abs}, \textit{ele-emptyset\_abs}, \textit{ele-ele\_abs})$$
$$= \textit{emptyset-ele\_abs}(a, \mathbf{r}, t, \textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{emptyset()}, \textit{ele}(\mathbf{r}, n)\rangle)) \in C[\langle \textit{emptyset()}, \textit{ele}(\{a\}\cup\mathbf{r}, n)\rangle]$$

$$\textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\{a\}\cup\mathbf{r}, n), \textit{emptyset()}\rangle, \textit{emptyset\_val}, \textit{emptyset-ele\_abs}, \textit{ele-emptyset\_abs}, \textit{ele-ele\_abs})$$
$$= \textit{ele-emptyset\_abs}(a, \mathbf{r}, t, \textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\mathbf{r}, n), \textit{emptyset()}\rangle)) \in C[\langle \textit{ele}(\{a\}\cup\mathbf{r}, n), \textit{emptyset()}\rangle]$$

$$\begin{aligned}
&\textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\{a_1\}\cup\mathbf{r_1}, t_1), \textit{ele}(\{a_2\}\cup\mathbf{r_2}, t_2)\rangle, \textit{emptyset_val}, \textit{emptyset-ele_abs}, \textit{ele-emptyset_abs}, \textit{ele-ele_abs})\\
&= \textit{ele-ele_abs}(a_1, \mathbf{r_1}, t_1, a_2, \mathbf{r_2}, t_2,\\
&\qquad \textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\mathbf{r_1}, t_1), \textit{ele}(\{a_2\}\cup\mathbf{r_2}, t_2)\rangle, \textit{emptyset_val}, \textit{emptyset-ele_abs}, \textit{ele-emptyset_abs}, \textit{ele-ele_abs}),\\
&\qquad \textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\{a_1\}\cup\mathbf{r_1}, t_1), \textit{ele}(\mathbf{r_2}, t_2)\rangle, \textit{emptyset_val}, \textit{emptyset-ele_abs}, \textit{ele-emptyset_abs}, \textit{ele-ele_abs}),\\
&\qquad \textit{Product(Set1(A),Set1(A))-elim}(\langle \textit{ele}(\mathbf{r_1}, t_1), \textit{ele}(\mathbf{r_2}, t_2)\rangle, \textit{emptyset_val}, \textit{emptyset-ele_abs}, \textit{ele-emptyset_abs}, \textit{ele-ele_abs})\\
&\qquad)\\
&\in C[\langle \textit{ele}(\{a_1\}\cup\mathbf{r_1}, t_1), \textit{ele}(\{a_2\}\cup\mathbf{r_2}, t_2)\rangle]
\end{aligned}$$

Figure 5*:* Set Product Computation Rules

4.2.3.4 Type Product Algorithm for Inductive Types

The type product algorithm for inductive types is similar to the one for recursive type constructors. However, because a data constructor with multiple variables of MVA type must perform induction over all of them simultaneously (see p. 65), the induction variables are associated with the data constructor and not the induction variables. Thus a premise can have at most three induction variables. This simplifies the algorithm which only allows for induction over the first, second, or both data constructor(s) in the premise where two inductive data constructors are crossed. This results in the final algorithm:

1. Find the recursive introduction variables in the introduction rules. Also, find the introduction rules which use multivalued attributes, and keep track of their base cases.
2. Create the elimination rule. The type premise and major premise are similar for all elimination rules we will consider. There is a minor premise for each introduction rule.

   The elimination rule creates a form which is used to break apart elements of the types into their constituents. The manner in which it is done is specified by the arguments to the form. The form is given an element of the type as its first argument. The other arguments tell how to eliminate the top level data constructor (as specified by the computation rules). There is one parameter for each data constructor and thus for each introduction rule. The elimination form has six kinds of parameters:

   (a) *self* — the first parameter is an element of the type. This is the expression which is to be reduced to a canonical form as specified by the computation rules.

   (b) VALue parameters — the value to be returned if the first argument is a constructor with no parameters.

   (c) Non-recursive, non-inductive abstractions — a lambda abstraction that will be applied if *self* is bound to a constructor which has no recursive introduction variables. Each parameter in the abstraction is bound to an argument in the data constructor.

   (d) Recursive abstractions — a lambda abstraction which will be applied if *self* is bound to a constructor which has recursive introduction variables. First, the arguments of the data constructor are given as the first parameters of abstraction. Then, recurse variables are created for the abstraction which correspond to the recursive introduction variables.

If this is a product type constructor, then an additional recurse variable must be created for the paired recursive introduction variable from each of the two constituent data constructors.

(e) Inductive abstractions — a lambda abstraction which will be applied if *self* is bound to a constructor which has introduction variables of type MVA. First, the arguments of the data constructor are given as parameters of abstraction. Then, inductive variables are created for the abstraction which correspond to the MVA introduction variables.

If this is a product type constructor and both data constructors contain MVA variables, then three induction variables must be created to correspond to induction over the first, second, and both data constructors.

(f) Recursive and inductive abstractions — Steps (2d) and (2e) are both executed.

3. Now, a computation rule is created for each introduction rule. There are four kinds of computation rules: one if a constructor has a recursive introduction variable, one if it has an induction introduction variable, one if it has both, and one if it has neither.

(a) no recursive or induction introduction variables — replace the major premise of the elimination rule with the premises of the introduction rule.

(b) with recursive introduction variable — replace the major premise of the elimination rule with the premises of the introduction rule as was done above. The right hand side of the equality in the conclusion is a call to the recursive abstraction with each recursing parameter bound to a (lazy) call of the elimination function where the first argument is the value bound to the associated recursive introduction variable.

(c) with induction introduction variable — replace the major premise of the elimination rule with the premises of the introduction rule as was done above. The right hand side of the equality in the conclusion is a call to the induction abstraction with the induction parameter(s) bound to a (lazy) call of the elimination function where the first argument is the value bound to a construction of the data constructor on the remaining elements of MVA form.

(d) with recursive and induction introduction variables — the computation is set up so that the induction occurs first, then the constructor may be recursed on. If the recurse variable is evaluated first, then the current induction hypothesis is lost. The FeatureStructure (section 6.2) type constructor contains a data constructor which fits this condition.

4. Then, a SPIDER function is created which lazily evaluates expressions of the type when given the appropriate WEB graph.

4.3 Operational Semantics

There are two approaches to using constructive type theory in program development. One approach is to develop a theorem proving system which would generate proofs based on the abstract specification. Then, because they are constructive proofs, it is possible to extract programs which will execute the function. This is the approach taken in NuPrl [CAB$^+$86] and Isabelle [Pau89].

We take a different approach. It is easier to write programs than proofs of correctness. It is usually easier to check theorems than to prove them. By implementing the inference rules of constructive type theory, we create a deterministic proof procedure. This allows the user to write a program, which when executed, will create a proper and appropriate sequence of inference rule applications (a proof). The user can then check the sequence to verify a proof of correctness. This could be used to automatically check an abstract specification, but because we are more interested in knowledge based applications than program verification, we do not use it in this manner. Instead, we use constructive type theory to give an operational semantics to functions written in SPIDER.

First we give two examples of using constructive type theory to prove program correctness and then give the semantics of SPIDER. It would not be difficult to abstract these proofs of correctness if the abstract specification is set up appropriately. However, as hash been shown in automated reasoning, setting up the appropriate specification can be difficult.

4.3.1 Proofs in Constructive Type Theory

Consider the function `nullp` from LIST(A) to BOOLEAN which returns `true` iff its one argument is `nil`. Thus we write its type description and abstract specification as:

$$nullp : List(A) \rightarrow Boolean$$

$$nullp(x) = \begin{cases} true, & \text{iff } x = nil; \\ false, & \text{otherwise.} \end{cases}$$

Now we write a SPIDER program which we hope fits the specification:

```
defsfun nullp Boolean(A)
```

```
        ()
      null() => true()
      cons(?a,?l) => false()
```

The defsfun form in SPIDER translates this to the lambda expression:[13]

$$nullp \equiv \lambda self.List\text{-}elim(self, true, \lambda a.\lambda l.\lambda rec\_l.false)$$

Now, using the elimination and computation rules for LIST(A) created by SPIDER, we can prove that this meets the abstract specification for `nullp`.

Proposition

$$\lambda self.List\text{-}elim(self, true, \lambda a.\lambda l.\lambda rec\_l.false) \;=\; \begin{cases} true, & \text{iff } self = nil; \\ false, & otherwise. \end{cases}$$

Proof:

Case 1: $self = nil$

$(\lambda self.List\text{-}elim(self, true, \lambda a.\lambda l.\lambda rec\_l.false))\ nil$

$\rightarrow List\text{-}elim(nil, true, \lambda a.\lambda l.\lambda rec\_l.false)$ by $\beta - reduction$

$\rightarrow true$ by nil-computation

Case 2: $self = cons(a, l)$ where $a \in A,\ l \in$ LIST(A)

$(\lambda self.List\text{-}elim(self, true, \lambda a.\lambda l.\lambda rec\_l.false))\ cons(a, l)$

$\rightarrow List\text{-}elim(cons(a, l), true, \lambda a.\lambda l.\lambda rec\_l.false)$ by $\beta - reduction$

$\rightarrow (\lambda a.\lambda l.\lambda rec\_l.false)\ a\ l\ List\text{-}elim(l, true, \lambda a.\lambda l.\lambda rec\_l.false)$ by cons-computation

$\rightarrow false$

These are the only two cases because there are no other introduction rules for LIST(A). ■

Now, consider the slightly more complicated function `member`, with type description, abstract specification, and SPIDER function definition of:

$$member : List(A) \times A \rightarrow Boolean$$

$$member(l, x) \;=\; \begin{cases} false, & \text{if } l = nil; \\ true, & \text{if } l = cons(y, l') \text{ and either } x = y \text{ or } member(l', y) = true; \\ false, & \text{otherwise} \end{cases}$$

```
defsfun member List(Symbol)
        (?ele)
  null() => false()
  pair(?first,?rest) => if eq ?first ?ele
                             true()
                           recurse()
```

The defsfun form translates to

$$member \equiv \lambda self.\lambda ele.List\text{-}elim(self, false, \lambda first.\lambda rest.\lambda rec\_rest.\ if\ \ ele\ \ eq\ \ first\ \ (true)\ \ h)$$

[13] The expression is implemented as a Common Lisp function.

Proposition

$$\lambda self.\lambda ele.\mathit{List\text{-}elim}(self, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ \ ele\ \ eq\ \ \mathit{first}\ \ (\mathit{true})\ \ rec\_rest)$$

$$= \begin{cases} \mathit{false}, & \text{if } l = nil; \\ \mathit{true}, & \text{if } l = cons(y, l') \text{ and either } x = y \text{ or } member(l', y) = \mathit{true}; \\ \mathit{false}, & \text{otherwise} \end{cases}$$

Proof:

Case 1: $self = nil$

$\lambda self.\lambda ele.\mathit{List\text{-}elim}(self, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ ele\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ nil\ \_$

$\rightarrow^* \mathit{false}$ *by nil-computation*

Case 2: $self = cons(a, l)$ where $a \in A,\ l \in List(A)$

By induction on "length" of list.

Base Case: $l = nil$

$\lambda self.\lambda ele.\mathit{List\text{-}elim}(self, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ ele\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ cons(a, nil)\ x$

$\rightarrow \mathit{List\text{-}elim}(cons(a, nil), \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow (\lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ a\ nil$

$\mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$ by pair-computation

$\rightarrow \mathit{if}\ x\ eq\ a\ (\mathit{true})\ \mathit{List\text{-}elim}(nil, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

Now, if $x = a$, then the condition is true, and we have

$\mathit{if}\ \mathit{true}\ (\mathit{true})\ \mathit{List\text{-}elim}(nil, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow \mathit{true}$ by Boolean-elim, technically by true-computation

Or, if $x \neq a$, then it is

$\mathit{if}\ \mathit{false}\ (\mathit{true})\ \mathit{List\text{-}elim}(nil, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow \mathit{List\text{-}elim}(nil, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$ by Boolean-elim

$\rightarrow \mathit{false}$ by nil-computation

In either case, the abstract specification is satisfied.

Induction Step:

Let n be the "length" of l. Consider the list of length $n + 1$.

This can only be formed by $cons(y, l)$, where either $y = x$ or $y \neq x$.

Case a: Show $member(cons(x, l), x) = \mathit{true}$

$\lambda self.\lambda ele.\mathit{List\text{-}elim}(self, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ ele\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ cons(x, l)\ x$

$\rightarrow \mathit{List\text{-}elim}(cons(x, nil), \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow (\lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ x\ l$

$\mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

by pair-computation

$\rightarrow \mathit{if}\ x\ eq\ x\ (\mathit{true})\ \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow \mathit{if}\ \mathit{true}\ (\mathit{true})\ \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

by definition of equality

$\rightarrow \mathit{true}$ by Boolean-elim

This satisfies the abstract specification.

Case b: Show $member(cons(y, l), x) = \mathit{true}$ iff $member(l, x) = \mathit{true}$, where $y \neq x$.

This is equivalent to showing that $member(cons(y, l), x) = member(l, x)$.

$\lambda self.\lambda ele.\mathit{List\text{-}elim}(self, \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ ele\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)\ cons(y, l)\ x$

$\rightarrow \mathit{List\text{-}elim}(cons(y, l), \mathit{false}, \lambda \mathit{first}.\lambda rest.\lambda rec\_rest.\ \mathit{if}\ x\ eq\ \mathit{first}\ (\mathit{true})\ rec\_rest)$

$\rightarrow (\lambda \mathit{first}.\lambda \mathit{rest}.\lambda \mathit{rec\_rest}.\ \mathit{if}\ x\ \mathit{eq}\ \mathit{first}\ (\mathit{true})\ \mathit{rec\_rest})\ y\ l$

$\quad\quad \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda \mathit{rest}.\lambda \mathit{rec\_rest}.\ \mathit{if}\ x\ \mathit{eq}\ \mathit{first}\ (\mathit{true})\ \mathit{rec\_rest})$

by pair-computation

$\rightarrow \mathit{if}\ x\ \mathit{eq}\ y\ (\mathit{true})\ \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda \mathit{rest}.\lambda \mathit{rec\_rest}.\ \mathit{if}\ x\ \mathit{eq}\ \mathit{first}\ (\mathit{true})\ \mathit{rec\_rest})$

$\rightarrow \mathit{if}\ \mathit{false}\ (\mathit{true})\ \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda \mathit{rest}.\lambda \mathit{rec\_rest}.\ \mathit{if}\ x\ \mathit{eq}\ \mathit{first}\ (\mathit{true})\ \mathit{rec\_rest})$

$\rightarrow \mathit{List\text{-}elim}(l, \mathit{false}, \lambda \mathit{first}.\lambda \mathit{rest}.\lambda \mathit{rec\_rest}.\ \mathit{if}\ x\ \mathit{eq}\ \mathit{first}\ (\mathit{true})\ \mathit{rec\_rest})$

by false-computation

which is equivalent to $\mathit{member}(l, x)$.

This covers the abstract specification. ■

4.3.2 Semantics for SPIDER

We give the semantics for the SPIDER forms `defstype` and `defsfun`. The SPIDER runtime environment consists of a collection of types $\mathcal{T}$, functions $\mathcal{F}$, and inference rules $\mathcal{I}$.

4.3.3 Type Definition

The syntax of `defstype` is:

$$\langle \mathit{defstype\ form} \rangle ::= \underline{\texttt{defstype}}\ \langle \mathit{name} \rangle\ (\langle \mathit{type\ par} \rangle^{+})\ \langle \mathit{scons\ def} \rangle^{+}$$

$$\langle \mathit{scons\ def} \rangle ::= \langle \mathit{scons} \rangle\ \langle \mathit{par\ type} \rangle\ = \ \langle \mathit{wcons\ name} \rangle \langle \mathit{con\ key\ forms} \rangle^{*}$$

$$\langle \mathit{type\ par} \rangle ::= \langle \mathit{variable} \rangle$$

$$\langle \mathit{type\ spec} \rangle ::=$$

$$\langle \mathit{par\ type} \rangle ::= \text{type spec with no variables}$$

$$\langle \mathit{con\ key\ forms} \rangle ::= \underline{\texttt{: BASE - CASE}}\ \langle \mathit{scons} \rangle$$

The semantics of TransDef$[\![\langle \mathit{defstype\ form} \rangle]\!]$

1. Add $\langle \mathit{name} \rangle(\langle \mathit{type\ par} \rangle^{*})$ to $\mathcal{T}$.
2. Create a new formation rule $\langle \mathit{name} \rangle$*-formation*

   $\langle \mathit{name} \rangle$**-formation**

   $$\frac{\langle \mathit{type\ par} \rangle_1\ \mathit{type}}{\langle \mathit{name} \rangle(\langle \mathit{type\ par} \rangle_1, \ldots)\ \mathit{type}}$$

   add this to $\mathcal{I}$.
3. For each $\langle \mathit{scons\ def} \rangle$,

a. Add to $\mathcal{I}$ an introduction rule.

b. Add the function to $\mathcal{F}$.

4. Create an elimination rule for the type using the algorithms of section 4.2 and the inference rules created above. Add this to $\mathcal{I}$.
5. For each introduction rule, create a computation rule and add it to $\mathcal{I}$.
6. Create a form for the elimination rule and add it to $\mathcal{F}$.

4.3.4 Function Definition

The `defsfun` form is defined by:

```
defsfun <name> <type> <args>
   <pattern> => <expr>
   <pattern> => <expr>
   <pattern> => <expr> ...
```

where the $\langle pattern \rangle$'s are sufficient to cover the types (as explained below).

$$
\begin{array}{rcl}
\langle type \rangle & ::= & \langle SPIDER\ type \rangle \ \ | \ \ \underline{(}\ \langle SPIDER\ type \rangle^{n \geq 1}\ \underline{)} \\
\langle args \rangle & ::= & \underline{(}\ \langle variable \rangle^{*}\ \underline{)} \\
\langle pattern \rangle & ::= & \langle constructor\ expr \rangle^{n}\ [\ \underline{\texttt{OR}}\ \langle constructor\ expr \rangle^{n}\ ]^{*}\ [\ \langle where\ clause \rangle\ ] \\
 & & \text{— } same\ n \text{ as } \langle type \rangle \\
\langle constructor\ expr \rangle & ::= & \langle constructor \rangle \\
 & | & \underline{\{}\ \langle constructor \rangle\ [\ \underline{::}\ \langle induction\ var \rangle\ ]\ \underline{\}} \\
\langle where\ clause \rangle & ::= & (\ \underline{\texttt{where}}\ \langle constraint \rangle^{+}\ \underline{\Rightarrow}\ \langle expr \rangle)^{+}\ \underline{\texttt{otherwise}} \\
\langle constraint \rangle & ::= & \langle variable \rangle\ \underline{\texttt{eq}}\ \langle variable \rangle \\
 & | & \langle variable \rangle\ \underline{\texttt{neq}}\ \langle variable \rangle \\
 & | & \langle variable \rangle\ \underline{\texttt{in remaining}}\ \langle variable \rangle \\
 & | & \langle variable \rangle\ \underline{\texttt{notin remaining}}\ \langle variable \rangle \\
\langle induction\ var \rangle & ::= & \langle variable \rangle
\end{array}
$$

A SPIDER expression is defined by:

$$
\begin{aligned}
\langle expr \rangle ::= {} & \underline{\texttt{LET}}\ [\ \langle variable \rangle\ \underline{\equiv}\ \langle expr \rangle\ ]^*\ \underline{\texttt{IN}}\ \langle expr \rangle \\
\mid {} & \underline{\texttt{CASE}}\ \langle variable \rangle\ \underline{\texttt{OF}}\ [\ \langle pattern \rangle\ \underline{\Rightarrow}\ \langle expr \rangle\ ]^+ \\
\mid {} & \langle variable \rangle \\
\mid {} & \langle fun\ call \rangle \\
\mid {} & \langle recurse\ form \rangle \\
\langle fun\ call \rangle ::= {} & \langle constant \rangle\ \underline{(}\ \langle expr \rangle^*\ \underline{)} \\
\langle recurse\ form \rangle ::= {} & \underline{(}\ \underline{\texttt{RECURSE}}\underline{)} \\
\mid {} & \underline{\texttt{RECURSE}}\ \underline{(}\ \langle recursive\ intro\ variable \rangle\ \underline{)} \\
\mid {} & \underline{\texttt{RECURSE}}\ \underline{(}\ \langle induction\ var \rangle\ \underline{)} \\
\mid {} & \underline{\texttt{RECURSE}}\ \underline{(}\ \langle recursive\ intro\ variable \rangle\ \langle recursive\ intro\ variable \rangle\ \underline{)} \\
\mid {} & \underline{\texttt{RECURSE}}\ \underline{(}\ \langle induction\ var \rangle\ \langle induction\ var \rangle\ \underline{)}
\end{aligned}
$$

where the variables in a recurse expression are either induction variables or recursive introduction variables (but not both). The form

$$RECURSE(\ \langle recursive\ intro\ variable \rangle\ \langle recursive\ intro\ variable \rangle\ )$$

can only occur in a function on a PRODUCT TYPE, and the two recursive introduction variables must come one from each half of the product.

The semantics of $\langle$*defsfun form*$\rangle$ are TransDef$[\![\langle defsfun\ form \rangle]\!]$ which adds the function to the collection of SPIDER functions $\mathcal{F}$ by translating it into a lambda expression in a manner similar to [Jon87].

Now, we examine how a function on an inductive type is defined. Consider the `member` function for SET(A) in SPIDER.

```
defsfun member Set(A)
     (?x)
  empty-set() => false()
  ele(?ele,?set)::?next
    => if ?x eq ?ele
          true()
        recurse(?next)
```

The induction variable `?next` in the function definition corresponds to the induction variable i in the inference rules.

The `defsfun` form is expanded into the lambda expression:

$$member \equiv \lambda self.\lambda x.Set\text{-}elim(self, false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.if\ \ x\ \ eq\ \ ele\ \ true()\ \ next)$$

This is used in evaluation as follows.[14]

Evaluate $member(ele(a, ele(b, ele(c, id()))), b)$

$= \ (\lambda self.\lambda x.Set\text{-}elim(self, false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.if\ x\ eq\ ele\ \ true()\ \ next))$
$\quad ele(\{a\} \cup (\{b\} \cup (\{c\} \cup \emptyset)), id_1) \quad b$

$\rightarrow Set\text{-}elim(ele(\{a\} \cup (\{b\} \cup (\{c\} \cup \emptyset)), id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.)$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by β-reduction

$\rightarrow (\lambda ele.\lambda ele\_rest.\lambda next.\lambda set.if\ \ b\ \ eq\ \ ele\ \ true()\ \ next) \quad a \quad (\{b\} \cup (\{c\} \cup \emptyset))$
$\quad Set\text{-}elim(ele(\{b\} \cup (\{c\} \cup \emptyset), id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by ele-computation

$\rightarrow if\ \ b\ \ eq\ \ a\ \ true()\ \ (Set\text{-}elim(ele(\{b\} \cup (\{c\} \cup \emptyset), id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by β-reduction

$\rightarrow^* Set\text{-}elim(ele(\{b\} \cup (\{c\} \cup \emptyset), id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by defn of *if* (Boolean-elimination), equality

$\rightarrow (\lambda ele.\lambda ele\_rest.\lambda next.\lambda set.if\ \ b\ \ eq\ \ ele\ \ true()\ \ next) \quad b \quad (\{c\} \cup \emptyset)$
$\quad Set\text{-}elim(ele(\{c\} \cup \emptyset, id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by ele-computation

$\rightarrow if\ \ b\ \ eq\ \ b\ \ true()\ \ (Set\text{-}elim(ele(\{c\} \cup \emptyset, id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by β-reduction

$\rightarrow if\ \ true\ \ true()\ \ (Set\text{-}elim(ele(\{c\} \cup \emptyset, id_1), false(), \lambda ele.\lambda ele\_rest.\lambda next.\lambda set.$
$\qquad if\ \ b\ \ eq\ \ ele\ \ true()\ \ next)$

by equality

$\rightarrow true()$ by Boolean-elimination

These are the same steps taken by the SPIDER evaluator. SPIDER can also be used to define more useful functions, such as intersection.

```
defsfun intersection <Set(A),Set(A)>
     ()
  empty-set()  empty-set()  =>  empty-set()
  empty-set()  ele(?ele,?set)::?ignore  OR
  ele(?ele,?set)::?ignore  empty-set()  =>  empty-set()
  ele(?x1,?set1)::?next1   ele(?x2,?set2)::?next2
    where ?x1 eq ?x2 => ele(?x1,recurse(?next1,?next2))
    where ?x1 in remaining ?x2 =>
         ;; the value of ?x1 occurs again as some value of ?x2
         ele(?x1,recurse(?next1))
    otherwise => ;; ?x1 not in the rest of ?x2
         recurse(?next1)
```

[14] We assume the order for creation of the MVA sets to be where the elements are stored in the reverse order they are added to the MVA set. This is just for exposition: evaluation does not depend on this order.

4.4 Inheritance

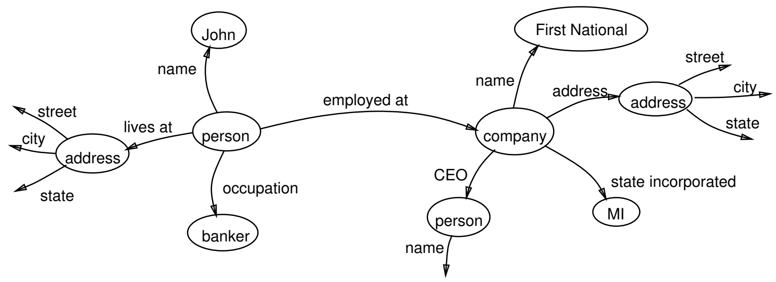
Another advantage of extensibility from using constructive type theory is that we can separate structure from type information. Structural information can be defined in WEB. This can be associated with the type information from SPIDER. However, the association need *not* be one-to-one. The same knowledge base structure can be associated with multiple types. This is usually avoided in strongly-typed systems, but by using constructive type theory we can reason with the type without making its structure explicit.

The advantage of this type/structure separation in a knowledge base is that we can define knowledge using one type and access it using another type. For example, we can define knowledge using a terminological subsumption language [BL85], then view it as a feature structure [KR86, Car92] and manipulate it using feature structure unification.

By having two languages, we place the structure information into WEB and the type information into SPIDER. This simplifies development and can lead to more effective knowledge sharing and a cleaner notion of polymorphism. WEB is a simpler and more efficient language, because there is no run-time type checking. The type checking takes place in SPIDER, where it can be done at compile time. Inclusion polymorphism [CW85] occurs when overlapping WEB graph primitives are used to define SPIDER types, specifically when one (or more) primitives in WEB are used in defining different data constructors for different types in SPIDER. This overlapping of primitives lets one structure have multiple types. The different SPIDER types can exploit the overlapping structure with polymorphic operations. This would not be possible if the type information were kept in WEB.

This separation between the primitives which define the knowledge base (WEB) and the type information which enforces type-correct reasoning (SPIDER) allows WEB to factor out polymorphism from SPIDER types. WEB makes explicit the common structure of polymorphic types. The WEB constructor which builds that structure is used to define all the SPIDER data constructors which are best described by that structure. Thus, there is an explicit link between the polymorphism of SPIDER types and the structure of that type (in WEB).

Consider the concrete example sketched below:



This information is represented by the SPIDER types Person, Company, and Address, which have very simple data constructors which merely create a new node or add one arc to the graph. Person has data constructors with type descriptors:

new-person:	() → Person
name:	Person × Symbol → Person
occupation:	Person × Symbol → Person
address:	Person × Address → Person
employed at:	Person × Company → Person

which would be used as in `name(occupation(new-person(), BANKER), JOHN)`. Company has data constructors with type descriptors:

new-company:	() → Company
name:	Company × Symbol → Company
ceo:	Company × Person → Company
address:	Company × Address → Company
state incorporated:	Company × Symbol → Company

Address is similar with data constructors for `street`, `city`, and `state`.

There is substantial overlap between the types Person and Company which could be used in a new common supertype NamedEntity with data constructors for `name` and `address`, which would incorporate the overlapping WEB graph primitives.

The three cases of overlapping primitives are:

1. The distinct data constructors have identical WEB graph structures.
2. The graph structure of one data constructor is a proper subset of the graph structure of another; i.e., the second constructor is strictly more specific than the first.
3. Both graph structures have some distinct, non-common, data constructors.

In addition, for any SPIDER type with multiple data constructors, each data constructor may fit into one of the three categories. Thus, two overlapping *types* may be identical, subtype-supertype, or partially overlapping. A type is a subtype of another iff any overlapping data structures are either identical or the graph structure of the data constructor of the subtype is a subset of the supertype's corresponding graph structure.

If one type is a subtype of another, we can allow methods to be inherited. This is represented in constructive type theory by adding a new *subsumption* inference rule. If Θ_1 is a subtype of Θ_2, we add the type subsumption inference rule Θ_1**-subsumption** to the inference rules of Θ_2.

Θ_1**-subsumption**

$$\frac{x \in \Theta_1}{x \in \Theta_2}$$

If two types partially overlap, we can create a common subtype which may have methods defined on it. This occurs when one or more data constructors of one type, say Θ_1, either occur in a second type, say Θ_2, or they subsume/are-subsumed-by data constructors in the type Θ_2. In the first case, let $\theta_1, \ldots, \theta_n$ be data constructors in Θ_1, and $\theta'_1, \ldots, \theta'_n$ be corresponding data constructors in Θ_2 which have the same graph constructor.[15] We can create a new type Θ_3 with data constructors $\theta_1, \ldots, \theta_n$ and replace them in Θ_1 with the *data constructor subsumption* inference rules:

θ_1**-subsumption** ... θ_n**-subsumption**

$$\frac{\theta_1(x_1, \ldots, x_{|\theta_1|}) \in \Theta_3}{\theta_1(x_1, \ldots, x_{|\theta_1|}) \in \Theta_1} \quad \cdots \quad \frac{\theta_n(x_1, \ldots, x_{|\theta_n|}) \in \Theta_3}{\theta_n(x_1, \ldots, x_{|\theta_n|}) \in \Theta_1}$$

where $|\theta_i|$ denotes the arity of the data constructor θ_i. We also add to the definition of Θ_2 the data constructor subsumption inference rules:

θ_1**-subsumption** ... θ_n**-subsumption**

$$\frac{\theta_1(x_1, \ldots, x_{|\theta_1|}) \in \Theta_3}{\theta'_1(x_1, \ldots, x_{|\theta_1|}) \in \Theta_2} \quad \cdots \quad \frac{\theta_n(x_1, \ldots, x_{|\theta_n|}) \in \Theta_3}{\theta'_n(x_1, \ldots, x_{|\theta_n|}) \in \Theta_2}$$

If two SPIDER types are structurally identical, then it is possible to form a *type union* over them and share all operations on them.

[15] It may be that θ_i and θ'_i have the same name. They must have identical arity.

4.4.1 Type Inclusion

Recall in the definition of SET1 (section 4.2.2.2) we noticed `new-id` and `ele(∅,(new-id))` are identical except for type information. Because of this a new type subsumption inference rule is to be added to the SET1 definition in SPIDER:

Ids-subsumption

$$\frac{n \in Ids}{n \in Set1}$$

This *type subsumption rule* is implemented in the inductive rule construction algorithm (section 4.2) by including the introduction inference rule(s) of IDS in the definition of SET1. The **Ids-subsumption** rule tells SPIDER that all the members of IDS are included in SET1, but SPIDER must also be told that these new members are equivalent to some old ones, namely that *new-id()* $=$ *ele*$(\emptyset,$ *new-id()*$)$. This occurs in the type SET1, and thus the full congruence rule is:

ele-base-equality

$$\frac{\text{new-id}() = t \;\in\; Ids}{\text{ele}(\emptyset, \text{new-id}()) = t \;\in\; Set1}$$

This results in an elimination rule with an added premise for the **Ids-subsumption** rule.

This gives an elimination rule of:

Set1-elimination

$$\frac{\begin{array}{l} [\![w \in Set1(A) \quad \triangleright \quad C[w]\ type]\!] \\ x \in Set1(A) \\ [\![n \in Ids \quad \triangleright \quad n \in Set1]\!] \\ [\![\text{new-id}() = n \;\in\; Ids \quad \triangleright \quad \text{ele}(\emptyset, \text{new-id}()) = n \;\in\; Set1]\!] \\ b \in C[\text{new-id}()] \\ [\![\;\; a \in A \\ \quad \mathbf{r} \subseteq A \\ \quad i \in C[\mathbf{r}] \\ \quad n \in Ids \\ \quad \triangleright \quad z(a, \mathbf{r}, i, n) \;\in\; C[\text{ele}(\{a\} \cup \mathbf{r}, n)] \\ ]\!] \end{array}}{\textit{Set1-elim}(x, b, z) \;\in\; C[x]}$$

Chapter 5
Application to Computational Genetics

I haven't any memory — have you?—
Of ever coming to the place again
To see if the birds lived the first night through,
And so at last to learn to use their wings.
— Robert Frost

We have developed a process for designing application-specific data models and have implemented it in WEAVE. We give our process for designing application-specific knowledge bases and describe its preliminary result when applied to integrating heterogeneous genome maps.

To design an application-specific data model using our approach, the knowledge base developers begin with a graphical sketch which appears to capture the structure and semantics required for the application. They use this to abstract common features of the sketch, then use WEAVE to group the abstractions into new data types. Methods are then developed to do reasoning on the data types, and the types and methods are collected to form a new data model. Any step can be repeated to refine the data model, and WEAVE is used to develop the knowledge base. We demonstrate this process on the problem of representing distance and order information for heterogeneous genome maps.

We have developed a general process for designing knowledge bases which can be used for heterogeneous genome maps. This process is supported by the strong, theoretical foundation given in previous chapters and is implemented by WEAVE. We demonstrate the process on a simple representation for distance and explain how queries can be asked on the knowledge base. We also show how a representation for order information can be developed in a similar fashion.

5.1 Genome Mapping

Mapping is the process of estimating the relative position of genes and other genetic markers on a chromosome and ascertaining the distance between them. Markers have a physical locations on a chromosome which can be identified by some laboratory procedure and whose pattern of inheritance can be followed. A genome map can be used to find the location of a specific gene whose location is not known by using laboratory procedures to discover which markers on the map are close to the gene in question. There are several different mapping processes and strategies with several resulting maps. In this paper, we deal with three different maps: genetic linkage maps, physical maps, and radiation hybrid maps.

Genetic linkage maps are based on the inheritance of genes and markers from one generation to another [Ott91]. Alternative forms of a marker (alleles) are studied within a pedigree (family) to determine their pattern of inheritance. Multiple markers can be examined, and statistical methods can be used to estimate the likelihood that they are linked, that is, close together on the same chromosome. Distance can be measured as the expected number of recombination events (crossovers) which occur between markers. This distance is measured in Morgans with 1 Morgan corresponding to one expected crossover per meiosis.

Physical maps vary in their degree of resolution depending upon the laboratory procedure used. They measure physical distance between markers in terms of the number of base pairs between two markers; the actual distance can only be estimated based on the resolution of the specific laboratory procedure used.

Radiation hybrid maps [CBP+90] are created by using a high dose of x-rays to break a human chromosome into several fragments. Laboratory procedures can be used to collect fragments into rodent-human hybrid clones which are analyzed for the presence or absence of specific markers. Each hybrid contains a sample of human fragments and statistical methods can be used to estimate the probability of a radiation-induced break between two markers. It appears that the frequency of breakage between markers is directly proportional to physical distance, and this distance can be recovered using statistical methods which take the possibility of multiple, intervening breakpoints into account. The distance is measured in Rays with 1 Ray corresponding to one expected break. Radiation hybrid maps attempt to measure physical distance (as do physical maps), but do this by breaking the chromosome at random locations, which requires statistical methods to recover distance (as is needed for genetic maps).

5.2 Genome Mapping Problem

One thing a geneticist wants from a database is to integrate the different kinds of genome maps. It needs to answer queries, such as:

What is the distance between markers?
Is there support for one order over another?
What is the consistency between marker orders?

WEAVE makes it easier to develop knowledge bases which answer these kinds of queries. The heterogeneous genome mapping problem is in need of direct knowledge base support. Currently, there are many different kinds of genome maps at different levels of granularity, with different properties, and with different ways in which they are useful. Each map is based on laboratory procedures which can have errors and inconsistencies. Different statistical methods are used to deal with the problems, and they are based on different assumptions and models. People can generally deal with one kind of map at a time, though it is tedious. When multiple, heterogeneous maps are available, it can be difficult to handle the complexity.

5.3 Knowledge Base Design Process

Often the best way to solve a problem is to change the way the problem is viewed [And85]. This requires a change in the representation of the problem state. However, most representation schemes require that a problem be represented in only one way. This can lead to a more efficient implementation, but requires a human to mentally coerce their reasoning process into a fixed, unnatural form (while trying to solve a difficult problem).

The solution to this problem is to have one formalism to represent the structure of the knowledge in a computationally effective form and let the user view the data in the manner most natural to the solution of the problem. If the user is unsure of the most natural representation, it is also important that the system be both flexible and extensible.

We have applied this to the problem of knowledge base design and have implemented a tool which can be used to develop knowledge bases that allow for multiple views of the same structure. This is done by allowing distinct data types to share a common structure for the data and is implemented via the layered architecture of WEAVE.

Integrating heterogeneous maps is an especially good problem on which to demonstrate this approach because there is already an underlying structure (the genome) which people view in different ways (physical and genetic maps). This is not to say that the most computationally efficient way of representing the underlying structure of the maps will correspond to the genome, but merely indicates that there is a common structure to the maps, and this can guide development toward a more effective implementation. It gives us a place to start and fixes the user's view to be the heterogeneous maps. This results in the goal to find a common structure which can be efficiently used to integrate the information contained in multiple, heterogeneous maps.

The knowledge base design process is:

1. **Create a graphical sketch.** This should capture the structure and semantics for the application.
2. **Abstract common features of the sketch.** These are sections of the graph that can be used to build and manipulate the graph in a meaningful way. They are specified in the graph description language WEB.
3. **Group the abstractions into data types.** These graph abstractions become data constructors for the type.
4. **Implement methods on the type.** These are implemented in the strongly-typed functional programming language SPIDER.
5. **Collect the types and methods to form a data model.** This forms the data model for the application's knowledge base.

In WEAVE it is possible to have multiple, overlapping type definitions on the same structure. This allows data to be entered using one view of the structure and retrieved using alternative views.

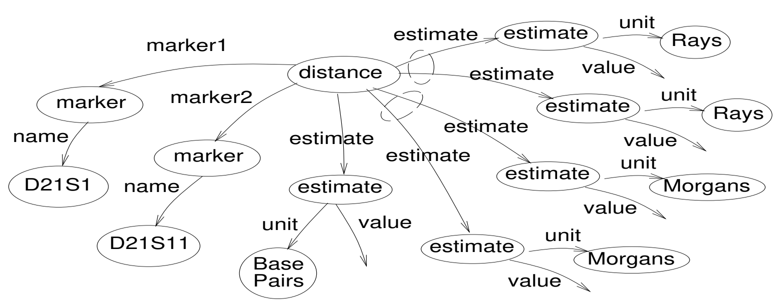
We now show how one view can be created for DISTANCE. This is demonstrated in terms of putting data into the knowledge base, although the same types are also used for retrieval. The same process is also used to develop overlapping types for retrieving the data.

5.4 Distance

Distance between markers in a genome map represents: expected number of recombination events (crossovers) between them, expected number of breaks induced by irradiation, or physical distance expressed in base pairs. Each of these distances can be estimated by a laboratory procedure. We give a common representation for the distances and their estimates, develop data types for them, and show how they can be combined to integrate distances from heterogeneous maps.

5.4.1 Abstracting Common Features

Distance between markers in a map can be represented graphically as a *distance* node with estimates of the distance represented as values of a *multi-valued attribute* (set-valued role) labeled *estimate*. These estimates should be thought of as being collected by the *units* of the distance estimates. For example, the distance between the markers D21S1 and D21S11 may be represented as



where the estimates are defined by multiple data sets.

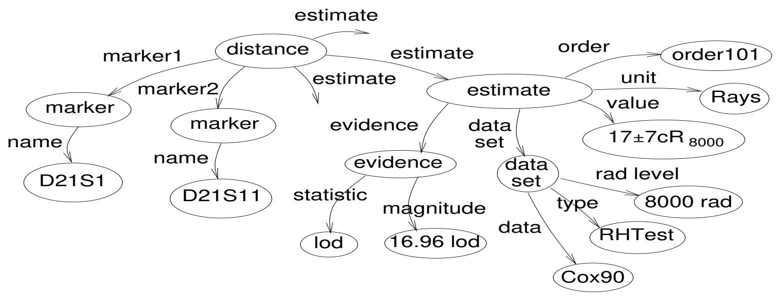
From this sketch an abstraction for distance can be formed in WEB which will construct the graph for distance.

$$\begin{aligned} wDistance(?marker1, ?marker2, ?estimate) \;\equiv\; & [\text{create } ?distance] \\ & (marker1\ \ ?distance\ \ ?marker1)\ (marker2\ \ ?distance\ \ ?marker2) \\ & (estimate\ \ ?distance\ \ ?estimate)\ [\text{return } ?distance] \end{aligned}$$

where *?name* denotes a variable, and we use $w\langle Constructor\rangle$ to make clear that this is a program defined in WEB. The abstractions in WEB that construct graphs in the knowledge base are called *graph constructors*.

The graph constructor *wDistance* is then associated with a data constructor for the data type DISTANCE. The data constructors are embedded in SPIDER and are used to build the knowledge base.

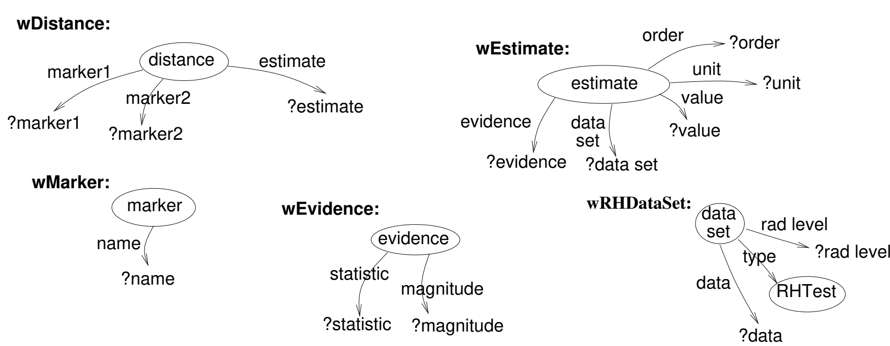
Much more information is needed in a representation of the estimate, such as the data set used, the order which the distance is based on, and the statistical evidence for the estimate. When these are included, it results in a representation such as:



which represents the distance information between D21S1 and D21S11 from a radiation hybrid data set [CBP+90].

Abstractions, data constructors, and data types can be generated from this sketch as follows. First, find the sections of the graph which are likely to be reused in a semantically meaningful manner. In this example, the concepts involved are: distance, marker, estimate, evidence, and data set. Each of these concepts are associated with a section of the graph. We then define a graph constructor to build each section (as was done for *distance* above). For this example, it is fairly straightforward to do, though WEAVE also has the more extensive capabilities which can also deal with more complex constructs, such as cyclic graphs, collections of multi-valued attributes, and indirection.

When we separate the sections of the graph, we are left with five graph constructors: *wDistance*, *wMarker*, *wEstimate*, *wEvidence*, and *wRHDataSet* which build the graphs



These five graph constructors have arguments as follows:

$$wDistance(?marker1, ?marker2, ?estimate)$$

$$wMarker(?name)$$

$$wEstimate(?value, ?unit, ?dataset, ?evidence, ?order)$$

$wEvidence(?statistic, ?magnitude)$

$wRHDataSet(?radlevel, ?data)$

Note that in this example the *value* includes both the estimate and a measure of variability.

5.4.2 Forming Data Types

Each of these graph constructors is associated with a data constructor for a user-defined type. The data constructor's parameters are typed and accessed through SPIDER. The graph is created when the data constructor is evaluated within SPIDER. The data constructors have type specifications:

$$distance : Marker \times Marker \times Estimate \rightarrow Distance$$

$$marker : Symbol \rightarrow Marker$$

$$estimate : Number \times Unit \times DataSet \times Evidence \times Order \rightarrow Estimate$$

where the types are created using the `defstype` form in SPIDER. The `defstype` form for DISTANCE is:

```
defstype Distance ()
  distance(Marker, Marker, Estimate) = wDistance
```

Most types have more than one data constructor. The common data type LIST has data constructors `null()` and `pair(?x,?l)`, and the type BINARYTREE has data constructors `leaf(?a)` and `node(?left,?right)`. For the current example, we have found it useful to have multiple data constructors for DATASET — for both a radiation hybrid data set, which takes the radiation level as another argument. and data sets that do not need that additional argument. Thus for DATASET, there are data constructors `wDataSet` and `wRHDataSet` with a type definition of

```
defstype DataSet (A)
  dataset(DataSetType,A) = wDataSet
  rad-dataset(Number,A) = wRHDataSet
```

The DATASETTYPE would be either *Genetic* or *Physical* in this example, as is shown in the next section. These data constructors can be used to build the knowledge base. The graph above can be built by the expression

```
distance(marker('D21S1),
         marker('D21S11),
         estimate(0.17, Rays,
```

```
rad-dataset(8000, 'Cox90),
evidence(lod, 16.96),
order(...)    -- as explained below
))
```

Functions are defined on the data types, and their execution is specified by a collection of inference rules in constructive type theory. These functions correspond to methods in object-oriented programming. These inference rules have traditionally been used for type inference or automated reasoning [BCM88, CAB$^+$86], but we use them to give an operational semantics to functions which operate on elements of the type in chapter 4. Some inference rules tell how to form the type and data constructors. For DISTANCE, these look like

Distance-formation

$$\frac{}{\mathit{Distance\ type}}$$

distance-introduction

$$\frac{m1 \in \mathit{Distance} \qquad m2 \in \mathit{Distance} \qquad e \in \mathit{Estimate}}{\mathit{distance}(m1, m2, e) \in \mathit{Distance}}$$

These inference rules are calculated in a straightforward manner from the `defstype` form above. We also have developed an algorithm as part of WEAVE which will calculate inference rules that tell how to eliminate a type into its constituents and perform computations on the type. This is described in chapter 4. These rules are then used to create a form in SPIDER which performs well-founded computations, i.e., under certain liberal conditions the computation can be guaranteed to halt.[16] If this proves overly restrictive for some application, a full (recursively enumerable), functional programming language , such as Lisp or SML [MTHM90], is also available for the cases where it is necessary.

The elimination and computation rules for DISTANCE are somewhat more complex and are given in appendix C. The details of the rules are not important for the understanding of how they are used. The elimination rule describes a function called `Distance-elim` which takes three arguments. The first is the expression to be evaluated. The others are lambda expressions that give the body of the function which is to be applied to the expression depending upon what the outermost data constructor of the expression is. The computation rules tell how the second and third arguments to `Distance-elim` are to be

[16] This occurs because all elements of a type must have been constructed through a finite (though unlimited) application of introduction inference rules. In addition, the functions on the type must be restricted to the primitive recursive functions.

used to calculate the result. This is translated into a lambda expression which is then evaluated using lazy evaluation when applied to a *distance*.

For example, a function to collect all the estimates of a distance into a list, regardless of the data set or units, could be defined in SPIDER as:

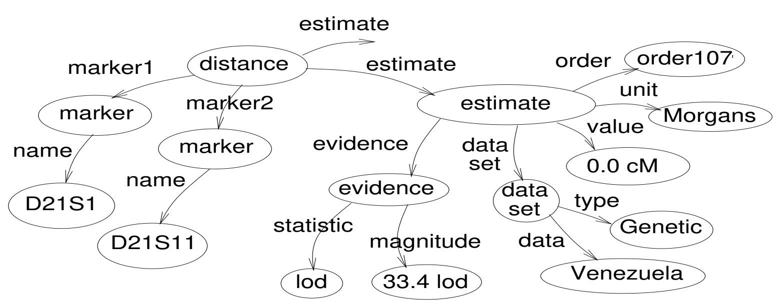
```
defsfun values (Distance) ()
  distance(?m1,?m2,empty) => nil()
  distance(?m1,?m2,?e)::?next
    => cons(estimate-value(?e), recurse(?next))
```

This is translated into the lambda expression:

$$\lambda x.\mathit{Distance\text{-}elim}(x, \mathit{nil}(), \lambda m1.\lambda m2.\lambda e.\lambda \mathbf{r}.\lambda i.\mathit{cons}(\mathit{estimate\text{-}value}(e), i))$$

5.4.3 Integrating Heterogeneous Maps

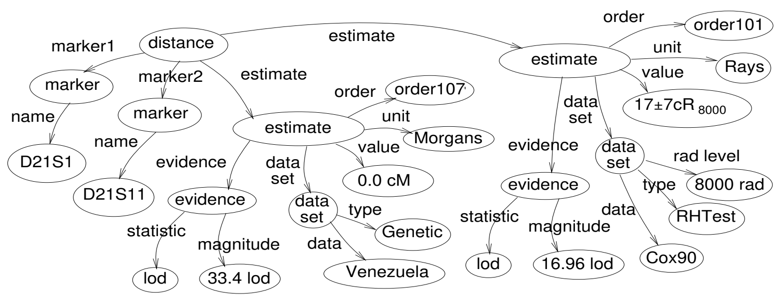
Although the process of defining a DISTANCE type was given for radiation hybrid mapping, a similar process can be used for other maps. The DISTANCE type can be used for genetic maps, and distance information can be shared between heterogeneous maps. For example, consider a representation of the distance between D21S1 and D21S11 from a genetic map taken from [THW+88]:



The same data constructors can be used in this instance as were used above for radiation hybrid distances.

```
D21S1 == marker('D21S1)
D21S11 == marker('D21S11)
VenDataSet == dataset(Genetic, 'Venezuela)
distance(D21S1,  D21S11,
         estimate(0.0, Morgans, VenDataSet,
                  evidence(lod, 33.4),
                  order(...)   -- as explained below
                  ))
```

This leads automatically to a combined graphical representation in the knowledge base:



Distance information from additional maps can also be added, and queries asking for specific information can be asked of the knowledge base.

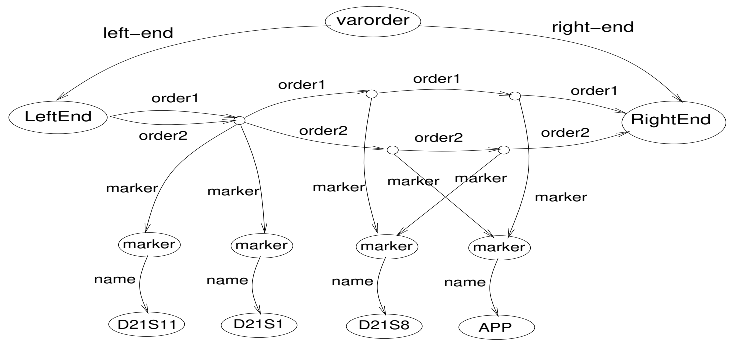
5.5 Order

Results similar to the types for DISTANCE can be obtained for order information. There are three dimensions of an order representation that we deal with here: intra-order uncertainty, inter-order uncertainty, and heterogeneity of maps. Intra-order uncertainty occurs when no order information is available for a collection of markers: they are either physically indistinguishable or tightly linked with no intervening crossovers or radiation breaks observed. Inter-order uncertainty occurs when markers can be distinguished, but there is still some uncertainty as to which is the actual order within the collection of markers and/or with respect to other markers, although one order may be more likely than another. Map order information can come from heterogeneous maps with different levels of granularity and sometimes conflicting orders.

To deal with this information we must represent:

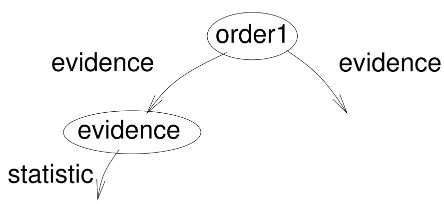
1. the order of markers and sets of markers,
2. a collection of orders which may be "partially ordered" by some likelihood statistic, and
3. collections of orders which may overlap in the markers ordered, but may conflict and have omitted data.

Although it is useful to have a simple way to represent known order for a collection of markers, it appears, for the general case, a representation is needed such as:



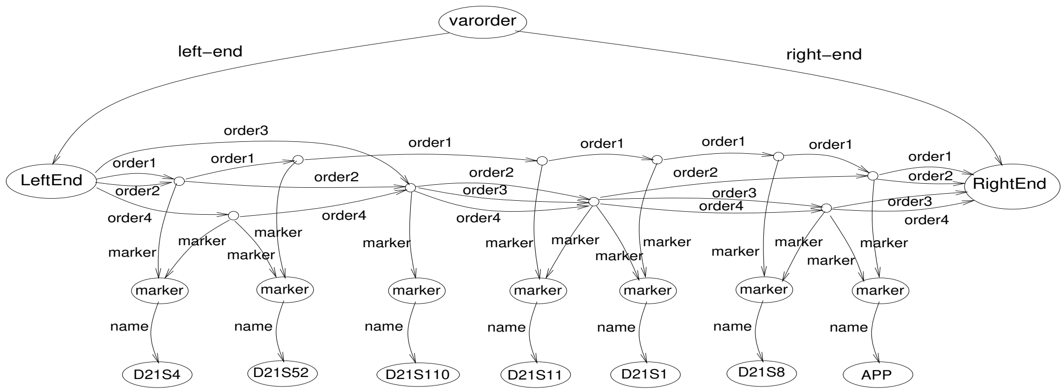
which represents uncertainty in the order D21S11, D21S1, D21S8, and APP from a genetic linkage map [WSL+89]. The markers D21S11 and D21S1 are tightly linked with no observed crossovers, and the order D21S11/D21S1 – D21S8 – APP is only 235 times more likely than D21S11/D21S1 – APP – D21S8 (usually not considered statistically significant because it is less than 10^3).

Abstractions, data constructors, and data types are then formed in a manner similar to the process for DISTANCE. In addition, the arcs in the graph, e.g., order1 and order2, can also be treated as nodes (thus WEB is a higher-order, binary logic programming language). This allows auxiliary information, such as statistical evidence, to be associated with an order. This is represented as:



and abstractions can be formed in like manner.

Order information from multiple, heterogeneous maps can be combined using the same data types. For example, we can enter order information from a physical map [GP92], a radiation hybrid map [CBP+90], and two genetic maps [WSL+89, TWS+92] of a portion of human chromosome 21. Each map is entered separately, but because of the overlap in markers, a knowledge base like the following is the result:



This represents the most likely order from each map. In this case, there are no conflicting orders, and a potential overall order can be obtained from WEAVE:

order num	map type	source	order
order1	Rad Hybrid	Cox 90	D21S4, D21S52, D21S11, D21S1, D21S8, APP
order2	Physical	Gardiner 92	D21S4, D21S110, $\substack{\text{D21S1}\\\text{D21S11}}$, APP
order3	Genetic	Warren 89	D21S110, $\substack{\text{D21S1}\\\text{D21S11}}$, $\substack{\text{APP}\\\text{D21S8}}$
order4	Genetic	Tanzi 92	$\substack{\text{D21S4}\\\text{D21S52}}$, D21S110, $\substack{\text{D21S1}\\\text{D21S11}}$, $\substack{\text{APP}\\\text{D21S8}}$
overall	—	—	D21S4, S52, S110, S11, S1, S8, APP

The overall order was obtained manually, but the process can be implemented using the topological sort algorithm [CLR90]. This would be developed as an part of an external problem solver or application. These external problem solvers and applications access WEAVE through the knowledge base manager and through its problem solver interface.

5.6 Knowledge Base Querying

One advantage of a knowledge base over an *ad hoc* system is the ability to query against it. Query processing is done in WEAVE through a simple knowledge base manager. Currently, the knowledge base manager is given a partially instantiated data constructor and retrieves the structures in the knowledge base which match it.

Although this is work in progress, we want to set the context in which knowledge base design is most useful. We are developing a natural language interface to the knowledge base manager which will allow for English queries to the knowledge base such as:

Find the distance between marker D21S1 and marker D21S11.

Find the best orderings.

Find order evidence for markers D21S16 and D21S48.

WEAVE is being used to implement the natural language interface, and this natural language interface application will also serve as another test and demonstration of WEAVE's effectiveness.

WEAVE can answer these queries and others like them now when expressed as data constructors such as:

```
distance(marker('D21S1), marker('D21S11), ?x)
?order in best-order(?dataset, ?order)
order(marker('D21S16), marker('D21S48), ?order)
```

The natural language queries and data constructor queries have a similar form which can be used in a unification-based natural language interface [Shi86]. The disadvantage in all implemented systems except ours is that this restricts the queries to have a form similar to the data constructors that were used to define the knowledge base.

5.7 Discussion

Although the simple queries mentioned above are interesting and useful in their own right, the real power of a natural language interface to a genome knowledge base occurs when reasoning methods can also be accessed through natural language. For example, topological sort could be accessed through the query:

Find the most likely overall order.

A query such as this should actually give several of the most likely orders along with the evidence used to rank them. More complex reasoners can also be included to deal with inconsistencies or cycles in the orders.

Another useful query would be:

What is the distance between D21S16 and D21S11?

which is not stored directly in the knowledge base, but which can be calculated using distance information between intervening markers (for some assumed order).

Another long-term goal would be to handle contingent or hypothetical queries:

What would be the distance between D21S1 and APP if the order
D21S1, D21S11, D21S18, D21S8, APP were assumed?

Queries such as these are not difficult to implement using WEAVE and would be very useful to molecular biologists.

In summary, there are several advantages to designing a knowledge base to represent heterogeneous mapping information.

1. A knowledge base organizes the information in a clear, integrated framework which allows inferences to be made more easily.
2. There is now a process for designing knowledge bases which can guide development and make more efficient use of the map maker's time.
3. The formalisms we have described here have proven themselves expressive enough for a wide variety of tasks and appear sufficiently powerful to help solve the problem of integrating heterogeneous maps.
4. Because these formalisms are very flexible yet can be implemented efficiently, they promise to be an effective tool for mapping the human genome.

Chapter 6
Other Applications

It is evident, then, that not everything demonstrable can be defined.
— *Aristotle*

This chapter contains application of our knowledge base design process to developing representation schemes for complex objects and feature structures from object-oriented databases and natural language semantics, respectively. We describe a simple constraint-based problem solver we have implemented and show how it uses an application-specific knowledge base to solve a logic puzzle. We then show how a natural language interface can be developed on a knowledge base developed using WEAVE.

6.1 Complex Objects

Complex objects are an inductive type. The type of COMPLEX OBJECT is defined as a generalization of SET. Instead of having elements of sets defined as values of the multi-valued attribute `ele`, elements of the COBJ type are formed as collections of labeled attribute-value pairs.

CObj-formation

$$\frac{A\ type}{CObj(A)\ type}$$

Complex object instances are built up from the data constructor `cco`.

cco-introduction

$$\frac{\mathbf{l} \subseteq Labels \quad \mathbf{v} \subseteq A \quad n \in Ids}{\mathrm{cco}(\mathbf{l}, \mathbf{v}, n) \ \in \ CObj(A)}$$

where the variables $\mathbf{l}$ and $\mathbf{v}$ are elements of MVA(LABELS) and MVA(A), $\mathbf{l}$ and $\mathbf{v}$ are constrained to be the same cardinality, and they are "paired". The LABEL type is a collection of labels (really, symbols). This pairing is accomplished in the implementation by requiring pairs to be defined individually. The MVA(A) sets are used only by the elimination rules. The elimination rule is:

CObj-elimination

$$\frac{\begin{array}{l}[\![w \in \mathit{CObj}(A) \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\ x \in \mathit{CObj}(A) \\ [\![\mathrm{id}() = n \ \in \ \mathit{Ids} \quad \triangleright \quad \mathrm{cco}(\emptyset, \emptyset, \mathrm{id}()) = n \ \in \ \mathit{CObj}(A)]\!] \\ b \in C[\mathrm{id}()] \\ [\![\ l \in \mathit{Labels} \\ \quad v \in A \\ \quad \mathbf{rl} \subseteq \mathit{Labels} \\ \quad \mathbf{rv} \subseteq A \\ \quad i \in C[\langle \mathbf{rl}, \mathbf{rv} \rangle] \\ \quad n \in \mathit{Ids} \\ \quad \triangleright \quad z(l, v, \mathbf{rl}, \mathbf{rv}, i, n) \ \in \ C[\mathrm{cco}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \\ ]\!]\end{array}}{\mathit{CObj\text{-}elim}(x, b, z) \ \in \ C[x]}$$

This elimination rule is created automatically by algorithms in SPIDER.

The new computation rules are:

$$\mathit{CObj\text{-}elim}(\mathrm{id}(), b, z) \ = \ b \ \in \ C[\mathrm{id}()]$$

$$\begin{aligned}\mathit{CObj\text{-}elim}(\mathrm{cco}(\{\mathrm{l}\} \cup \mathbf{rl}, \{\mathrm{v}\} \cup \mathbf{rv}, \mathrm{n}), \mathrm{b}, \mathrm{z}) \ &= \ z(l, r, \mathbf{rl}, \mathbf{rv}, \mathit{CObj\text{-}elim}(\mathrm{cco}(\mathbf{rl}, \mathbf{rv}, n), b, z), n) \\ &\in \ C[(\mathrm{cco}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)]\end{aligned}$$

The COBJ type constructor can be either combined with SET(A) to form non-first normal form relations [Hul87] or extended to be recursive by changing the `cco` form to be over the type $\mathit{MVA}(\mathit{Labels}) \times \mathit{MVA}(\mathit{CObj}(A)) \times \mathit{Ids}$.

The function `attr-value` can be defined on the type to return the value at a specific attribute.

```
defsfun attr-value CObj(A)
     (?x)
  id() => false()
  cco(?attr,?val,?ignore)::?next
    where ?x eq ?attr => ?val
    otherwise => recurse(?next)
```

This is equivalent to the lambda expression:

$$\begin{aligned}\mathit{attr\text{-}value} \equiv \lambda \mathit{self}.\lambda x.\mathit{CObj\text{-}elim}(\mathit{self}, \mathit{false}(), \lambda l.\lambda v.\lambda l\_rest.\lambda v\_rest.\lambda next.\lambda cobj. \\ \mathit{if}\ (x\ \mathit{eq}\ \mathit{attr})\ \mathit{val}\ \ \mathit{next})\end{aligned}$$

6.2 Feature Structures

Feature structures are defined using the two *data constructors* `empty-fs` and `cons-fs`. Each data constructor is associated with a knowledge base constructor which creates the appropriate entities and associations in the knowledge base. `Empty-fs` creates a new feature structure which has no features coming from it. `Cons-fs` takes as arguments a (new) feature, its value, and an existing feature structure, and then modifies the feature structure to have the appropriate feature-value. Features are created by a WEB program called `new-feature`.

Reasoning on feature structures is traditionally done by finding the most general unifier. Unification is defined by a *reasoning method*, `unify-fs`, written in the knowledge base programming language SPIDER.

Feature structures usually only allow for single-valued features. By developing them in WEB, it is easy to generalize them to multi-valued features. This allows aggregation (conjunctive sets) to be introduced implicitly, without the need for an additional construct [Rou91], thus leading to a simpler unification algorithm when combined with disjunctive sets.

Features structures are an inductive type. The type of FEATURESTRUCTURE (abbreviated FS) is defined as a generalization of SET. Instead of having elements of sets defined as values of the multi-valued attribute `ele`, elements of the FS type are formed as collections of labeled attribute-value pairs.

FS-formation

$$\frac{A \; type}{FS(A) \; type}$$

Feature structure instances are built up from the data constructor `cfs`.

cfs-introduction

$$\frac{\begin{array}{l}\mathbf{l} \subseteq Labels \\ \mathbf{v} \subseteq FS(A) \\ n \in Ids\end{array}}{\mathrm{cfs}(\mathbf{l}, \mathbf{v}, n) \;\in\; FS(A)}$$

where the variables $\mathbf{l}$ and $\mathbf{v}$ are elements of MVA(LABELS) and MVA(A), $\mathbf{l}$ and $\mathbf{v}$ are constrained to be the same cardinality, and they are "paired". The LABEL type is a collection of labels (symbols). The pairing is accomplished in the implementation by requiring pairs to be defined individually. The MVA(A) sets are used only by the elimination rules. The elimination rule is:

FS-elimination

$$\begin{array}{l}
[\![w \in FS(A) \quad \triangleright \quad C[w]\ type]\!] \\
x \in FS(A) \\
[\![\mathrm{id}() = n \ \in \ Ids \quad \triangleright \quad \mathrm{cfs}(\emptyset, \emptyset, \mathrm{id}()) = n \ \in \ FS(A)]\!] \\
b \in C[\mathrm{id}()] \\
[\![\ l \in Labels \\
\quad v \in FS(A) \\
\quad \mathbf{rl} \subseteq Labels \\
\quad \mathbf{rv} \subseteq FS(A) \\
\quad i \in C[\langle \mathbf{rl}, \mathbf{rv} \rangle] \\
\quad h \in C[v] \\
\quad n \in Ids \\
\quad \triangleright \quad z(l, v, \mathbf{rl}, \mathbf{rv}, i, h, n) \ \in \ C[\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \\
]\!] \\
\hline
FS\text{-}elim(x, b, z) \ \in \ C[x]
\end{array}$$

This elimination rule is created automatically by algorithms in SPIDER.

The new computation rules are:

$$FS\text{-}elim(\mathrm{id}(), b, z) \ = \ b \ \in \ C[\mathrm{id}()]$$

$$\begin{aligned}
&FS\text{-}elim(\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n), b, z) \\
&\qquad = \ z(l, r, \mathbf{rl}, \mathbf{rv}, FS\text{-}elim(\mathrm{cfs}(\mathbf{rl}, \mathbf{rv}, n), b, z), FS\text{-}elim(v, b, z), n) \\
&\quad \in \ C[(\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)]
\end{aligned}$$

The function `attr-value` can be defined on the type to return the value at a specific attribute.

```
defsfun attr-value FS(A)
     (?x)
  id() => false()
  cfs(?attr,?val,?ignore)::?next
     where ?x eq ?attr => ?val
     otherwise => recurse(?next)
```

This is equivalent to the lambda expression:

$$attr\text{-}value \equiv \lambda self.\lambda x.FS\text{-}elim(self, false(), \lambda l.\lambda v.\lambda l\_rest.\lambda v\_rest.\lambda next.\lambda fs. \\ \hfill if\ (x\ eq\ attr)\ \ val\ \ next)$$

6.3 Problem Solving

To demonstrate how knowledge base design can be used for general problem solving, we have developed a knowledge base which stores information necessary to solve a logic puzzle. We then show an intelligent problem solver can use the knowledge base and how the knowledge base supports solution with a simple constraint-based problem solver. This demonstrates that the representation is not tied to the problem solving strategy used. We also show the problems solving with two representations. The first one is based on frames, and the second one on feature structures.

Examine the Rock City logic puzzle from [Dal90].

> "The Rock City Boosters" is a dinner club made up of businesspeople in a small western city. They meet each week to plan ways to boost their city. There are five officers – president, vice president, secretary, treasurer, and recorder – who are, not necessarily respectively, an attorney, a baker, a banker, a grocer, and a realtor. From the following clues you should easily decide the name and occupation of each of the club's officers.
>
> 1. The president has been a member of the club for only two years. Bob believes the president was elected solely on account of financial position.
> 2. Sally has been a member for about five years.
> 3. The only formal item at the club dinners is the seating arrangement. The president sits at the center of the head table, flanked on the right hand by the vice president and on the other side by the secretary. Ray and the realtor sit in the other two seats.
> 4. The attorney is the most popular officer. No one doubts this accounted for his/her election.
> 5. The grocer has no one on the right hand, but Sam sits on the left.
> 6. The baker sits between Angie and Bob.
> 7. Sally and the recorder do not sit next to each other.

There are at least three ways that the problem can be solved using graph unification [Kni89, AK84, KR86]:

1. Brute force — as Prolog would do it.
2. Intelligent search — solve the problem using an external reasoner and/or some external knowledge. This method assumes something else is giving the answer. The given answer can then be checked to make sure it is valid (for example, using situation theory [BE90a] [BE90b]).
3. Constraint-based approach — consider each statement as a set of positive and negative constraints on the solution. If there is a decision, then evaluate them by cases. This is the "array" method [Jr.57]. This method might lend itself to analysis using situation theory, too.

WEAVE currently supports the three methods. Although method (3) is probably the most interesting problem solver, methods (1) and (2) best illustrates this system and are used below.

Most real-world problems involve several different types of information. Even semi-realistic "toy" problems often must address this issue. This puzzle has several types of information which must be represented. These different types of information involve many different representation issues. However, we can simplify the task by restricting the representation to the knowledge needed to solve the puzzle and by making use of applicable linguistic and puzzle-solving conventions. For this simple problem, the useful representations are frames, sets, and diagrams.

1. officer — a businessperson with an office. This can be represented by the frame:

   $\{officer$

   $name = \_\_\_\_$

   $occupation = \_\_\_\_$

   $office = \_\_\_\_\}$

2. name — one of Bob, Sally, Ray, Sam, or Angie which can be represented as the set $\{Bob, Sally, Ray, Sam, Angie\}$.
3. occupation — one of attorney, baker, banker, grocer, or realtor which can also be represented as a set.
4. office — one of president, vice-president, secretary, treasurer, or recorder which can also be represented as a set.
5. length of membership — a property of businessperson which gives the length of time they have been a member of the club. This can be represented as another slot of the businessperson frame.
6. reason elected — an officer can be elected to an office because of popularity or financial position. This can be represented as another property of an officer, and thus, as another slot of the officer frame:

   $\{officer$

   $name = \_\_\_\_$

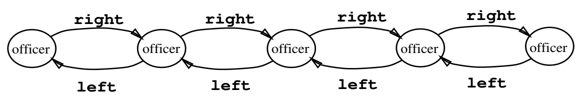
   $occupation = \_\_\_\_$

   $office = \_\_\_\_$

   $length\_membership = \_\_\_\_$

   $reason\_elected = \_\_\_\_\}$

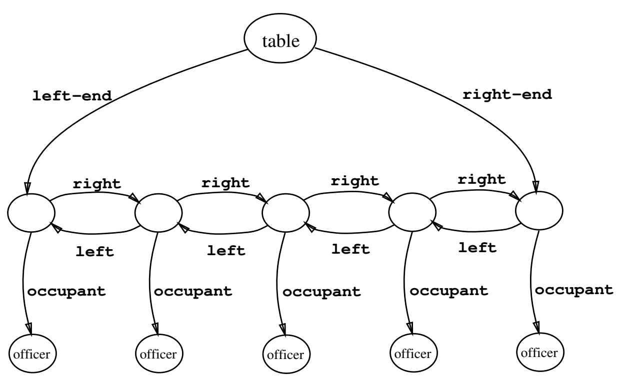
7. table — a left to right adjacency ordering of five officers. There are several ways to represent this, but given the structure of the information being modeled, a graphical representation would look something like:



6.3.1 Extending Types to Tables

Most of the information in the puzzle could be described by existing representation schemes (sets and frames), but the "table" could not. Because of the internal structure of the information, the most succinct description of the table is a graph to describe the "doubly-linked frames".

One way of defining the table is to use three types: TABLE, SEAT, and CHAIR. The table is composed of five adjacent seats, and each seat has a chair which is occupied by an officer. This can be graphed as:



The TABLE type consists of the "table" node and "left-end" and "right-end" arcs. There are five instances of the CHAIR type in the graph. Each is an unmarked node with an "occupant" arc pointing to an existing officer frame. The SEAT type is used to create the "left" and "right" arcs and is explained below.

The type TABLE is created by the form:

```
defstype sTable (A)
     table(sChair(A),sChair(A)) = wTable
```

which also creates a function `table` of type description

$$table : Chair(A) \times Chair(A) \rightarrow Table(A)$$

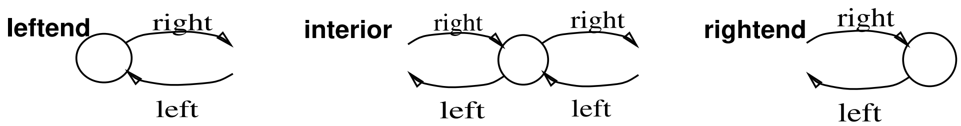
When `table` is called it creates the appropriate graph which is specified by the WEB graph constructor `wTable`:

$$wTable(?left\text{-}end, ?right\text{-}end) \equiv [\text{create } ?table]\ (left\text{-}end\ ?table\ \ ?left\text{-}end)\ (right\text{-}end\ ?table\ \ ?right\text{-}end)\ [\text{return } ?table]$$

This creates the "table" node and two arcs coming from it labeled "left-end" and "right-end" which point to the nodes given by the variables *?left-end* and *?right-end* respectively.

The CHAIR type is defined similarly.

There are two useful ways of defining the SEAT type. One way is to have one data constructor `adjacent` to specify that two chairs are adjacent. The second way is to have three data constructors `leftend`, `rightend`, and `interior` which create the arcs as follows:



The advantage of this system is that both definitions, SEAT1 and SEAT2, can be used without redundancy in the data. This occurs because overlapping (identical) graph primitives are used to define both types. The table seats can be defined using the first approach (which is the simplest) then accessed using the second approach (which provides more information to the problem solver).

Other access methods can then be defined on these types for easier problem solving. For example,

```
defsfun left-of Seat2(A)
        ()
  leftend(?seat,?right) => :ERROR
  interior(?left,?seat,?right) => ?left
  rightend(left,?seat) => ?left
```

which returns the seat to the left of the seat specified.

6.3.2 Validating a Solution Path

There are many ways in which the Rock City Puzzle may be solved. However, as the emphasis of this work is on representation, not problem solving or reasoning, we will discuss here only how WEAVE might be used by an external reasoner.

The reasoner is implemented in a traditional programming language and accesses functions created by SPIDER as any other function in the language. These functions are created by SPIDER using `defstype`, which creates the data constructors for the type, and `defsfun`, which creates functions on the type.

> "The Rock City Boosters" is a dinner club made up of businesspeople in a small western city. They meet each week to plan ways to boost their city. There are five officers – president, vice president, secretary, treasurer, and recorder – who are, not necessarily respectively, an attorney, a baker, a banker, a grocer, and a realtor. From the following clues you should easily decide the name and occupation of each of the club's officers. [17]

From the information in the introductory paragraph, the following definitions can be made:

For example, the officer frame is defined as:

```
officer :=[18]
   (frame OFFICER
          NAME        (set BOB SALLY RAY SAM ANGIE)
          OFFICE      (set PRESIDENT VICE-PRESIDENT ...)
          OCCUPATION  (set ATTORNEY BAKER BANKER ...))
```

An officer frame has three slots: name, office, and occupation. The value of each slot is a set representing its domain of possible values.

The individuals are defined similarly. The officer named "Bob" can be defined as:

```
bob := (frame OFFICER
              NAME BOB)
bob << officer
```

These definitions define the value of `bob` *to be a subclass of officer with the name specified to be "Bob", then give the frame the other values from officer. Thus,* `bob` *has value*

{*officer*

---

[17] The implemented system finds the correct solution as follows. However, the SPIDER code below has been changed syntactically to make the problem solving clearer. Also, the "set" function is not currently implemented as shown.

[18] This is a binding of the variable "officer" to a WEB graph, not an assignment, thus it does not prevent SPIDER from being functional, i.e., it is like `let`, not `setq`. However, an alternative interpretation would be to consider it as assignment. This would be similar to "references" in in SML [MTHM90] or object creation in Machiavelli [BO90].

$name = Bob$
$occupation = \{attorney, baker, banker, grocer, realtor\}$
$office = \{president, vice\text{-}president, secretary, treasurer, recorder\}\}$

```
sally := (frame OFFICER
              NAME SALLY)
sally << officer

ray := (frame OFFICER
              NAME RAY)
ray << officer
```

The definitions for Sam and Angie are similar.

Define the offices:

```
president := (frame OFFICER
                    OFFICE PRESIDENT)
president << officer

vice-president := (frame OFFICER
                         OFFICE VICE-PRESIDENT)
vice-president << officer
```

The definitions for secretary, treasurer, and recorder are similar.

Define the occupations:

```
attorney := (frame OFFICER
                   OCCUPATION ATTORNEY)
attorney << officer

baker := (frame OFFICER
                OCCUPATION BAKER)
baker << officer
```

The definitions for banker, grocer, and realtor are similar.

1. The president has been a member of the club for only two years. Bob believes the president was elected solely on account of financial position.

   1a. The president has been a member of the club for only two years.

   ```
   officer << (frame OFFICER
                     LENGTH-OF-MEMBERSHIP (set 2YRS 5YRS)))
   ```

   The length-of-membership property can have value of either 2 years or 5 years in this puzzle. The property is represented as a (new) slot on the officer frame, and it has as its value the domain of possible values represented as an (enumerated) set.

   ```
   president << (frame OFFICER
   ```

```
                    LENGTH-OF-MEMBERSHIP 2YRS)
```

1b. The president was elected solely on account of financial position. Ignore who believes it.

```
officer << (frame OFFICER
                    REASON-ELECTED (set FINANCIAL POPULARITY))

president << (frame OFFICER
                    REASON-ELECTED FINANCIAL)
```

1c. Ignore that Bob is not the president, for now. *This can be implemented by removing Bob from the set in the name slot of President and removing President from the set in the office slot of Bob.*

2. Sally has been a member for about five years.

```
sally << (frame OFFICER
              LENGTH-OF-MEMBERSHIP 5YRS)
```

3. The only formal item at the club dinners is the seating arrangement. The president sits at the center of the head table, flanked on the right hand by the vice president and on the other side by the secretary. Ray and the realtor sit in the other two seats.

   Make use of the additional information that Ray is on the extreme right. *This information would have to come from the external problem solver. For clarity, we do not use the* TABLE *type from the previous section, but delay its use until the next section.*

```
table1 := realtor
table2 := secretary
table3 := president
table4 := vice-president
table5 := ray
```

   These five table variables are then specified as entries in an instance of the "table" representation type. This allows the adjacency constraints later in the problem to be used.

4. The attorney is the most popular officer. No one doubts this accounted for his/her election.

```
attorney << (frame OFFICER
                  REASON-ELECTED POPULARITY)
```

5. The grocer has no one on the right hand, but Sam sits on the left.

   Thus, Table = ____ ____ ____ Sam grocer

```
table4 == sam
table5 == grocer
```

The variables `table4` *and* `sam` *are now constrained to be equivalent. They are both names for the unification of the values of* `table4` *and* `sam`*.*

6. The baker sits between Angie and Bob.

   Use order and position of: Bob, baker, and Angie with Bob at the extreme left. *This information comes from the external problem solver. It is a choice of one of six possible order/position combinations.*

   I.e., Table = Bob baker Angie ____ ____

   ```
   table1 == bob
   table2 == baker
   table3 == angie
   ```

7. Sally and the recorder do not sit next to each other.

   Only place for Sally to sit is Table2. *This follows from the partial solution currently available to the problem solver.*

   ```
   table2 == sally
   ```

   Recorder can only be in Table4 or Table5, but Table4 already has officer slot filled.

   ```
   table5 == recorder
   ```

Now, find the answer.

Bob is the only officer without an office. Treasurer is the only office left.

```
bob == treasurer
```

Sam and Angie do not have occupations. Attorney and banker are the only occupations left.

By brute force, Angie will not unify with attorney (because of reason-elected).

```
angie == banker
sam == attorney
```

This gives the answer:

```
<cl> (print-officers)

 OFFICER TOP
  OFFICE:                      PRESIDENT
  NAME:                        ANGIE
```

```
 OCCUPATION:                 BANKER
 LENGTH-OF-MEMBERSHIP:       2YRS
 REASON-ELECTED:             FINANCIAL
OFFICER TOP
 OFFICE:                     VICE-PRESIDENT
 NAME:                       SAM
 OCCUPATION:                 ATTORNEY
 REASON-ELECTED:             POPULARITY
OFFICER TOP
 OFFICE:                     SECRETARY
 NAME:                       SALLY
 OCCUPATION:                 BAKER
 LENGTH-OF-MEMBERSHIP:       5YRS
OFFICER TOP
 OCCUPATION:                 REALTOR
 NAME:                       BOB
 OFFICE:                     TREASURER
OFFICER TOP
 NAME:                       RAY
 OFFICE:                     RECORDER
 OCCUPATION:                 GROCER
```

6.3.3 A Simple Constraint-Based Problem Solver

We now show how a simple constraint-based problem solver can be used to solve the Rock City puzzle. The puzzle is set up as in the previous section through the first two steps, but here we use feature structures instead of frames. We set up the table as described by the TABLE type. We then set up a choice point in the problem solver as:

```
(choose (occup1 == realtor occup5 == Ray)
        (occup1 == Ray     occup5 == realtor))
```

where `occup`n refers to the n^{th} occupant of the table. The occupants can be referred to indirectly through the TABLE type, but we do not show that here.

4. The attorney is the most popular officer. No one doubts this accounted for his/her election.

   ```
   attorney << (create-fs 'REASON-ELECTED 'POPULARITY
                          (empty-fs))
   ```

5. The grocer has no one on the right hand, but Sam sits on the left.

   Thus, Table = ____ ____ ____ Sam grocer

   ```
   grocer == (chair-occup (table-right-end table))
   ```

```
Sam == occup4
```

The `occup4` variable can also be referred to as:

```
Sam == (left-of grocer)
```

6. The baker sits between Angie and Bob.

```
(choose (occup1 == angie occup2 == baker occup3 == bob)
        (occup1 == bob   occup2 == baker occup3 == angie)
        (occup2 == angie occup3 == baker occup4 == bob)
        (occup2 == bob   occup3 == baker occup4 == angie)
        (occup3 == angie occup4 == baker occup5 == bob)
        (occup3 == bob   occup4 == baker occup5 == angie))
```

This can also be implemented as:

```
(choose ((adj Angie baker)
         (adj baker Bob))
        ((adj Bob baker)
         (adj baker Angie)))
```

7. Sally and the recorder do not sit next to each other.

```
sally == (choose occup1 occup2 occup3 occup4 occup5)
recorder == occup5
```

The `recorder` variable can also be specified by:

```
recorder == (choose-set (non-adjacent-seats sally table))
```

Now, find the answer.

Bob is the only officer without an office. Treasurer is the only office left. This is discovered automatically when SET is used to define the possible officer names, occupations, and offices.

```
bob == treasurer
```

Sam and Angie do not have occupations. Attorney and banker are the only occupations left. By brute force, Angie will not unify with attorney (because of reason-elected).

```
(choose (Angie == attorney Sam   == banker)
        (Angie == banker   Sam   == attorney))
```

This gives the correct answer.

6.4 Natural Language Processing

We show how the natural language sentence:

Find the distance between marker D21S1 and marker D21S11.

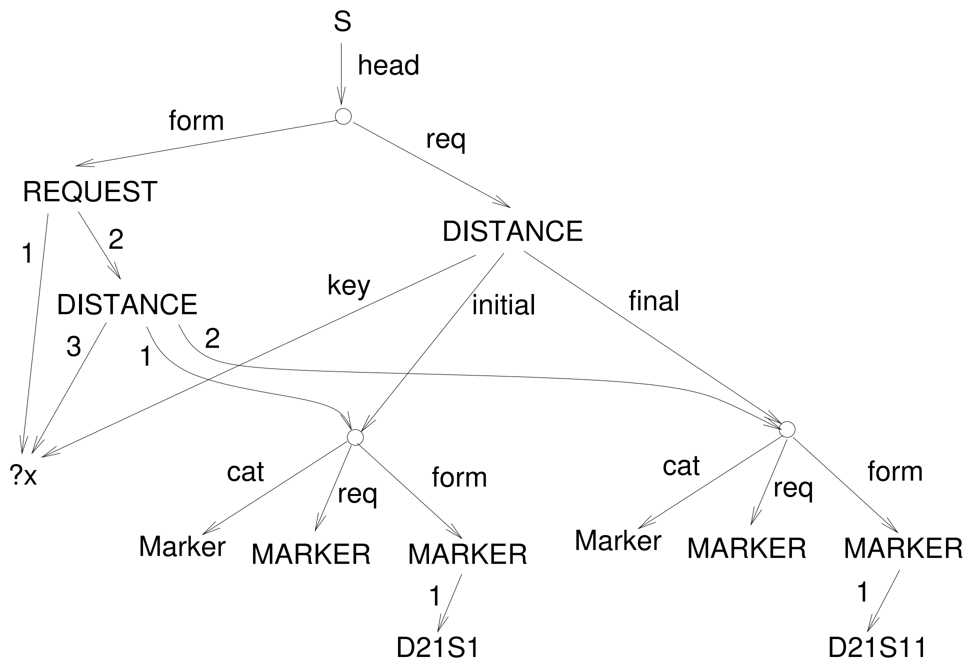
is parsed. This is answered by querying the WEAVE knowledge base manager (KBM)

```
?x in distance(marker('D21S1),marker('D21S11),?x)
```

The English query is parsed using a unification grammer [Shi86] which is modified to build up the KBM query as the sentence is parsed. The rules needed to parse this sentence are modified from a traditional semantic grammar, and we compact some levels of detail to make the exposition clearer.

```
 S -> "find" NP Range
     <S head req> = <NP head req>
     <S head req initial> = <Range head range initial>
     <S head req final> = <Range head range final>
     <S head req key> = new-variable()
     <S head form> = REQUEST
     <S head form 1> = <S head req key>
     <S head form 2> = <NP head form>
 Range -> "between" Marker_1 "and" Marker_2
     <Range head range initial> = <Marker_1 head>
     <Range head range final> = <Marker_2 head>
 Marker_1 -> "marker" Marker_2
     <Marker_1 head> = <Marker_2 head>
Word distance
     <head cat> = Noun
     <head req> = DISTANCE
     <head form> = DISTANCE
     <head form 1> = <head req initial>
     <head form 2> = <head req final>
     <head form 3> = <head req key>
Word D21S1
     <head cat> = Marker
     <head req> = MARKER
     <head form> = MARKER
     <head form 1> = 'D21S1
```

The request is then sent to the KBM by building up the structure contained in the `form` features. The simplified final parse structure for the sentence is:



This yields the form

```
REQUEST(?x, distance(marker('D21S1),marker('D21S11),?x))
```

which returns the correct distance.

Chapter 7
Related Work

What everybody echoes or in silence passes by as true today may turn out to be falsehood to-morrow, mere smoke of opinion, which some had trusted for a cloud that would sprinkle fertilizing rain on their fields.
— Henry David Thoreau

There is currently no system incorporating all of WEAVE's capabilities. Thus, there is no one system against which to compare WEAVE. There are, however, a variety of related approaches to representing knowledge. They tend not to be expressive enough to represent information naturally, but tend to coerce it into a framework which is alien to the domain.

The first system to attempt to represent semantic knowledge was Quillian's semantic networks [Qui68]. This was the first associative network formalism and is the precursor of current attribute value formalisms, conceptual modeling, and semantic databases. Semantic networks attempted to capture the associative properties of human cognition by linking closely related concepts. Reasoning could then be done through spreading activation, where links are followed to discover closely related concepts. This did not work in the way it was intended because uneven coverage of knowledge in the domain tended to bias the "distance". Closely related concepts in the area of interest were more distant than relatively unrelated concepts because more information (concepts and links) was added in the domain of interest than in peripheral areas. Semantic networks did, however, demonstrate their ability to represent static association and structure [Win70, Sch72, SGC79, Bra79, Fah79], but knowledge representation needs more than just semantic networks. We use SPIDER's type discipline to break up the huge graphs of semantic networks into small, understandable, WEB graphs which are associated with SPIDER data constructors.

Before implemented systems, there was the attempt by mathematicians and philosophers to use logic to represent information. Frege [Fre79] is credited with developing the first theory of first-order logic. It was an awkward language with only three primitives: implication, negation, and universal quantification. In 1883, Peirce independently developed a notation for logic based on Boolean algebra with "+" for disjunction, "·" for conjunction, "Σ" for existential quantification, and "Π" for universal quantification. These symbols were later changed by Peano to what we use today because he wanted to mix mathematical and logical quantifiers in the same formula. But in 1896, Peirce gave up on the linear notation and adopted a graphical notation for reasoning called existential graphs [Rob73]. These graphs have mechanisms for reasoning which are more expressive than the ones commonly used today. Although we use a different formal foundation for WEB, existential graphs are suggestive of a way to do rule-based inference in WEB graphs.

In 1975, Minsky [Min75] proposed the frame as a mechanism for representation. Over time, this was influenced by object-oriented programming [SB86] to become a record-oriented representation without the strong encapsulation or object identity [KC86] usually included in object-oriented systems but which sometimes included procedural daemons on the slots to increase the type of applications which could be supported.

Meanwhile, frames were being combined with semantic networks into a language called KL-ONE [Bra80]. This was expanded into a large family of knowledge representation languages called terminological subsumption languages derived from KL-ONE (discussed below) and formalized by Aït-Kaci [AK84].

In addition, work was being done to find a logical foundation for semantic networks [DK79, FH77], which overlapped the development of declarative programming languages [Kow79]. The logic for semantic networks could be viewed as a first order predicate calculus restricted to binary predicates.

Within artificial intelligence, this gives us three primary declarative representation schemes — frames, logic, and semantic nets — and various combinations of them. But within databases, researchers were trying to use the representation schemes to develop object-oriented [ZM90, Vos91], logic [GM78, DKM91, Zan90], and semantic databases [VB82, Abr74, HK87, PM88] as well as improve existing ones [MB89]. Programming language research has also been influenced by the representation schemes [BL87, AKN86, AKL88] and by their integration into databases [BB90, ZM90].

A portion of this work is geared toward trying to re-integrate research directions which although they had common ancestors, quickly lost touch with what was being done in other parts of the field. We re-examine the attempt to give semantic networks a firm, logical foundation and use theoretical techniques which were not available the first time. We then apply the result to the work which has been done on semantic databases and knowledge bases. We also apply techniques from constructive type theory to database programming languages and apply algebraic methods to data modeling. The result is a knowledge base design tool WEAVE which combines a semantic knowledge base WEB with a knowledge base programming language SPIDER and which has a rigorous theoretical foundation.

7.1 Attributive Description Formalisms

Currently, there are two predominant *attributive description formalisms* [NS90]. There are *terminological subsumption languages*, which are derived from KL-ONE [BS85], and *feature structures*, which evolved in computational linguistics [KR86, Car92, Shi86]. We propose a third attributive description formalism which uses a relational approach to define multi-valued attributes (similar to the binary roles of terminological subsumption languages), but which uses graph querying as the primary processing paradigm, rather than classification or graph unification as used in terminological subsumption languages or feature structures, respectively.

WEB has a graphical framework based upon semantic networks. It combines aspects of knowledge representation languages [MBJK90], feature structures [KR86, Car92], ψ-types [AK84] (which are a foundation for terminological subsumption languages [BS85]), semantic data models [HK87, PM88], and binary logic programming [DK79, BL87]. It also has aspects similar to Conceptual Graphs [Sow84], but organizes higher-order constructs differently. Most logic-based systems only consider first order predicate calculus as a logical foundation. WEB may be modeled as a higher-order predicate logic restricted to binary predicates. WEB uses graph querying for knowledge base access and does not do classification for terminological reasoning [BS85, BBMR89].

7.2 Binary Representation

WEB uses a binary logical formalism to represent semantic network-like structures. However, rather than implementing deductive inference procedures on the semantic network [DK79, FH77], we define reasoning methods in SPIDER and embed the methods in the networks using graph unification. This is similar to Restricted Binary Logic Programming [BL87], which also is oriented toward database retrieval, but which uses a data-driven model of computation. WEB is more expressive than Restricted Binary Logic because WEB is a higher-order logic.

The emphasis on binary predicates is an old one which showed the relationship between semantic nets and predicate logic then was quickly dropped in favor of n-ary predicates. However, there are two advantages in returning to binary logic for WEB. The first is that it forms a simple foundation which can be manipulated automatically. This is very important for extensibility. There is also not the original disadvantage of unwieldiness that led to the embrace of n-ary predicates because the user does not deal directly with binary logic but uses it only through SPIDER.

The second advantage is that it is easy to treat the binary predicates as attributes in semantic nets, roles in frames, arcs in graphs, etc. This allows the designer of the types in SPIDER a natural foundation upon which to develop application specific types.

Binary data models have been examined for semantic databases. One particularly similar data model to WEB is also one of the earliest: the *semantic binary data model* [Abr74] tried to have a minimalistic set of primitive constructs from which to build more powerful structures. This later led to the development of the NIAM (Nijssen Information Analysis Methodology) data model [VB82] which has influenced conceptual schemas in relational databases and led to the development of other binary data models [Mar83, Ris85, Ris86]. Binary formalisms have also been used in a graphical framework for other databases [PPT91, GPG90, CCM92]

7.3 Extensible Semantic Data Model

The development of a knowledge base design tool depends heavily on the data model for the underlying knowledge base. Because WEB is an attributive description language, the knowledge base programming language SPIDER is oriented toward traditional knowledge representation structures such as frames and semantic nets rather than addressing the integration of databases and extended predicate calculus derivatives [Zan90].[19] This is a generalization of work on feature structures and Aït-Kaci's ψ-types. Feature Structures can also model complex objects [BK86, KNN89, AFS89, Oho88, HK87, Heu89] and relational databases.

Brodie (in [Bro84]) proposes a family of semantic data models as *special purpose* or *application-oriented data models* which would capture the heterogeneous structure of complex data from areas such as: CAD/CAM databases, cartography, geometric shapes and figures, scientific applications, and VSLI. WEAVE does not have the diverse applicability of Brodie's proposal, but also does not restrict the new application-oriented data models to be similar to semantic models. Instead, the structure of the complex data is further abstracted (away from the semantic data model) and is described formally in terms of its own data model (a collection of SPIDER types).

WEB includes the persistence and querying facets from databases and some of the features of object-oriented or deductive databases. From object-oriented databases [SS91], WEB includes *data abstraction* to capture associations between items in the knowledge base, generalizations (taxonomic hierarchies), and aggregations (record structure). It includes *modularization* to provide encapsulation at a higher level of granularity. This supports belief revision, inconsistent knowledge, common knowledge, and multiple ontologies by separating incompatible groupings of knowledge. From deductive database features, we include inferencing, value identity to support extensional equality, negation, and a declarative query language. In addition, WEB supports disjunctive data for conditional reasoning and representing sets and incomplete knowledge. Circular definitions are also important for defining recursive concepts or describing common knowledge (where all agents are aware that all agents have that knowledge).

The semantic data models are the data models most similar to WEB. We have expanded on that idea using more recent techniques of modeling abstraction (section 7.3.1) and higher

[19] This paper will not discuss the tradeoffs with the logic approach to developing a knowledge base. It will instead merely emphasize formalisms that structure and organize data.

order concepts (section 7.3.2). We have also expanded on the notion of extensibility (section 7.3.3).

7.3.1 Abstractions

Semantic data models attempt to isolate the user from the structure of the data by introducing complex abstraction mechanisms. The four most common mechanisms are: aggregation (relations), associations (homogeneous sets), generalization (ISA hierarchy inheritance), and classification (class instantiation)[20] [PM88]. Most semantic data models also allow for non-normal (hierarchical) aggregations such as record structures. These mechanisms are known as type constructors in the semantic data modeling literature, but because our types are in SPIDER, not in WEB, they are more aptly described as data constructors in WEAVE.

Rather than defining a collection of built-in abstractions, WEB includes mechanisms to allow the user to define his or her own abstractions, as is done in Conceptual Graphs [Sow84]. These abstractions are defined declaratively by giving the relationship of attributes in the knowledge base. For example, the generalizations can be defined by defining the binary relation ISA. More complex abstractions are defined by giving a set of binary relations which must hold. The dynamic aspects of these relations are defined by graph querying.

7.3.2 Higher Order Constructs

In addition to defining attributes, WEB also allows for attributes of attributes, etc., and dynamic attributes whose value is calculated by some user-defined function. Besides these generalizations of attributes, WEB also defines generalizations of "entities" by allowing not only primitive entities, but encapsulated collections of entities and attributes and also dynamic entities whose reference may change. Encapsulated collections are useful for developing independent sub-knowledge bases which might contain contradictory knowledge. Dynamic entities are useful for modeling change parametrizing the knowledge base; this is especially useful for problem solving using hypothetical reasoning.

[20] This is different from the classification of terminological subsumption languages. There, classification creates subsumption relations between generalized concepts.

7.3.3 Extensibility

Higher level abstractions are defined in terms of graph primitives and other abstractions (as described above). This allows the semantic data model to be extended. The abstractions can then be encapsulated and associated with data constructors for newly-defined SPIDER types. Thus, we refer to WEB as an extensible semantic data model, though when used in conjunction with SPIDER, WEB is first extended to allow for natural representation of the application data, then restricted by access through SPIDER so that only the part of WEB which is needed for the application is actually available.

SPIDER restricts the structure of of WEB, abstracts it, and encapsulates it. This creates views of the knowledge base which may be thought of in terms of their own data model. This aspect of WEAVE is related to data model generation.

A *data model generator* creates data models to fit the requirements of specific applications [PM88]. Other data model generators are the Data Model Compiler [MH85, Mar86], EXODUS [CDF$^+$86, CDG$^+$90] and GENESIS [BBG$^+$88]. SPIDER differs from these systems by specifying the data models in terms of constructive type theory and then compiling the data model into an extended semantic data model (WEB). It also differs from current extensible data bases by being extensible at both the data type and the data model level.

An extensible knowledge base programming language must allow the knowledge base type system to be extended at multiple levels of granularity. It must include extensible types (i.e., object-oriented types which have a mechanism for defining sorts or classes). It must also allow for the addition of new types, such as those needed by temporal or spatial reasoners, application-specific types, or data types not predefined in the system, e.g., doubly-linked lists or binary trees. In addition, the knowledge base programming language must allow for new extensible types to be defined, such as frames with multiple inheritance [Car84], typed feature structures [Car92], or other kinds of types [Car88]. It is by the use of constructive type theory that this level of extensibility is obtained within a clean mechanism.

7.4 Knowledge Representation Languages

Telos [MBJK90] is a knowledge representation language designed to support the development of information systems. It is a specialized formal language but not a programming language ([MBJK90], p. 326). A knowledge representation language is intended to assign some (conceptual) "meaning" to statements in the language while a programming language deals with the data independently of its extensions. Although a knowledge base programming language is capable of describing much more complex data than traditional databases, it does not assign a built-in meaning to the statements using some (deductive) mechanism as is done in a knowledge representation language. Instead, the semantics is external to the knowledge base and is defined by the reasoners which use the knowledge base; these reasoners must insure that the data in the knowledge base is interpreted consistently.

For querying, Telos includes the commands `ask` and `retrieve`. (`Retrieve` is a simpler operation which does only limited inference.) Both commands allow for either proving that a closed formula follows from the knowledge base or finding propositions which will make a given open formula true (such as SQL allows on a relational database). SPIDER allows for `retrieve` on both closed and open formulas. The expensive `ask` operation is developed on top of SPIDER where it is tailored to the data model.

Propositions in Telos are organized along three dimensions referred to [HK87] as: *structured/aggregate* (record structures), *classification* (class instantiation), and *generalization* (ISA hierarchy inheritance). A knowledge base programming language must support aggregation and some kind of hierarchical modularization, though it is not clear it needs the built-in classes and inheritance of a knowledge representation language. SPIDER supports structured/aggregate, *grouping/association* (homogeneous sets), and a hierarchical partitioning mechanism. It does not have classes built-in; this allows for experimentation with inheritance.

It is not feasible to address all the knowledge representation issues, so the goal is to have a system whose knowledge model is more general than most (implemented) representation formalisms and use data types developed in constructive type theory [ML82, Bac86a] to restrict the expressiveness to what is needed for the particular application. This is similar to EpiKit [SG91] which accesses a *uniform structure* (KIF) with *heterogeneous inferencing*. However, EpiKit is an unstructured library of reasoning procedures while SPIDER organizes reasoning methods on a strongly-typed (multi-sorted) scheme. SPIDER and EpiKit also differ from *specialist* systems which use a *uniform interface* to *heterogeneous representations* such as Joshua [Shr91], K-Rep [MDW91], Josie [NBF91], Rhet [All91], CycL [LG90], or ECoNet [SPT87]. Joshua, K-Rep, and Josie differ from the other specialist systems by being extensible via a *protocol of inference*, which is a specific number of methods (usually object-oriented) that define the inference mechanisms. Because SPIDER accesses a uniform knowledge base, it does not need a protocol of inference. Instead, it uses user-defined inference methods to develop reasoners for the newly defined types.

The purpose of WEAVE is to design knowledge bases and not to improve the efficiency of automatic theorem provers as the hybrid reasoners or specialist systems do. The goal of specialist systems is to make the existing inference engine more tractable. SPIDER's goal is to add new inference mechanisms.

SNePS is based on propositional semantic networks and is geared toward natural language understanding. It requires the network to be very expressive and be able to reason with circular definitions and inconsistencies.

Most of the knowledge representation systems which have been developed have been based on terminological subsumption. Two other systems with similarities to our work are Algernon [CK91] which was designed to gain theoretical understanding of Access-Limited Logic [Cra89] and CAKE [Ric82, Ric85] which has a layered architecture.

7.4.1 Terminological Subsumption Languages

Terminological subsumption languages are based on the original work by Brachman [Bra80, BS82, BS85] to integrate frames and semantic networks.

KL-TWO [VM83, Vil85] and KRYPTON [BFL83a, BFL83b, BGL85, Pig84a, Pig84b] are hybrid knowledge representation systems which contain both a terminological reasoner (T-Box) and an assertional reasoner (A-Box). In KL-TWO, the assertional reasoner is a limited propositional reasoner [VM83] which is augmented with a terminological reasoner called NIKL (New Implementation of KL-one) [KMS83, PS89, KBR86]. NIKL defines roles as two-place relations. In KRYPTON, the assertional reasoner is a full first-order predicate logic.

BACK [NvL87, NvL88, Neb88] is a logically based hybrid knowledge representation system [vLPNS87] which emphasizes reasoning about instances.

KRIS [BH91] was developed as a prototype to gain theoretical understanding of hybrid terminological reasoning. It contains a sound and complete terminological reasoner

[BL84]. CLASSIC [PSMB+91, BBMR89, BMPS+90] emphasizes both practical application and theoretical understanding with its logical foundation being a (almost complete) terminological reasoner. Like BACK, KRIS, and CLASSIC, WEAVE emphasizes a theoretical understanding but is based on constructive type theory instead of a terminological logic.

ITL has a 2-level architecture which combines terminological knowledge with a Prolog-like relational reasoner [Gua91].

KRS [Gai91] is a cleanly designed tool with an efficient implementation.

KANDOR [PS84] is a small system.

MESON [OK89, EO86] unifies databases and knowledge representation languages by modifying the ABox to assume a unique name hypothesis and closed world assumption.

King Kong [BV91] was designed to be a subsystem for the natural language interface to a transportable database expert system. It is extensible and emphasizes relations as entities (not sets).

LiLog [BHS90, PvL90, Ple91] integrates ideas from the KL-ONE family and feature logic into order-sorted predicate logic. It also was developed for natural language applications.

LOOM [Mac88, MB87] extended classification and added backward chaining to terminological reasoning.

SB-ONE [AJWRR89, ARS90, All90, Kob91] was designed to add constructs which are needed for natural language processing such as sets and part-of relationships (with transitivity), and different types of defaults. Sets do appear essential for natural language processing [ARS90], thus a knowledge base programming language oriented toward natural language should allow for disjunctive data, though it is not necessary for set operations to be built-in. Disjunctive data are supported in WEAVE through multi-valued attributes.

WEAVE does not do classification of terms, but emphasizes knowledge base querying as shown in Chapter 2. Classification could be implemented using SPIDER, but this does not appear necessary.

7.4.2 Efficiency Concerns

Because of tradeoffs in expressiveness and tractability in representation languages, it is not possible to have a very expressive language with an efficient (tractable) reasoner. This has led to two opposing views:

1. Restrict the expressiveness of the language so a general-purpose reasoner is tractable (eg, SL-resolution over Horn Clauses (Prolog) or KL-ONE style languages).
2. Have an expressive language with an intractable (possibly incomplete) reasoner (e.g., a full theorem prover).

Some have argued for a compromise [Dav91] of:

3. A usually, but not always, fast reasoner that cannot always solve the problem, but is sufficient most of the time.

Others have argued that the emphasis is wrong. That we need:

4. An expressive language with specialized reasoners which can solve the common problems fast and a general reasoner to solve the less common problems more slowly. This is the hybrid reasoning approach.

We argue that the best approach is:

5. A language that expresses everything you want to express, but no more, so the reasoners are as fast as possible.

For this to occur, it must be possible for the user to define the expressiveness of the language. Rather than require the user to build up their own language (in a possibly *ad hoc* manner), we take the *restricted language* approach and allow the user to omit all constructs whose expressiveness is not needed.

7.5 Programming Languages

SPIDER is a simple, restricted programming language (as Abiteboul proposes in the declarative paradigm [Abi89]) and has the constructs of a simple, strongly-typed, functional programming language [CW85, Jon87]. Rather than develop a large inclusive language for accessing the knowledge base, such as Machiavelli [BO90] or E [CDG+90], SPIDER was developed to only perform knowledge base related tasks and to be embedded in a larger functional language which would be used to create the other application programs. SPIDER is used to develop both applications (like Machiavelli) and internal access methods (like E). This is why SPIDER is described as an *extensible* knowledge base programming language.

A knowledge base programming language will have certain *structural* and *behavioral* (functional) requirements in order to serve as an interface between knowledge-rich applications and a knowledge base. Structurally, it must contain association, taxonomic, and modularization constructs, and it must have a well-specified semantics. Behaviorally, it also must support both querying and reasoning.

To make these requirements more specific, we will look at an analogous database programming language, Machiavelli, and a programming language for natural language processing, LIFE [AKL88], to gain insight into how a knowledge base programming language should be developed.

The language E is part of the extensible database EXODUS. It consists of extensions to C++ which tend to deal with storage issues and query optimization more than data model extensions. Thus, it is not as similar to SPIDER as Machiavelli and LIFE.

Database programming languages (DBPLs) are not applicable for knowledge representation tasks because they are designed for large quantities of data with a highly repetitive structure and well-specified interactions. Knowledge base programming languages (KBPLs) are needed to handle data with a more varied structure and complex interactions, along with their possible extensions. For example, in databases, objects are *encapsulated* data structures, while in knowledge bases, frames, semantic nets, and feature structures are not; this allows for more complex structure and interactions at the expense of localizing behavior. Thus, DBPLs address issues of transaction management, access control, integrity, and resiliency, while a KBPL must address terminological reasoning, classification, consistency checking, common knowledge, and belief revision. For this, a KBPL must handle complex queries over varied structures, constraints, and modularized knowledge. Any Turing-expressive DBPL *can* handle these constructs, but they do not commonly do so.

Two useful properties for a KBPL are *strong-typing* and *extensibility*. Strong-typing modularizes and organizes the data, eliminates type errors (through static type checking), and can make reasoning more efficient, for example, by having specialized (efficient) reasoners which operate on some type (such as temporal relations). It is important that a KBPL be extensible so it can be oriented to specific problems without needing to be all-inclusive. For example, spatial reasoning may need specialized reasoning mechanisms for intersection and containment as well as specialized representations for points, lines, and polygons.

A KBPL, like a DBPL, must also address the *impedance mismatch* problem [AB87, BM88, ZM90] by cleanly integrating the knowledge base types and operations with the rest of the programming language. Specifically, the same types must be expressible in both the programming language and the database, and a programming language paradigm (declarative, object-oriented, functional, etc.) compatible with the database data model must be used. For example, object-oriented databases use an object-oriented data manipulation language to access the database to solve the impedance mismatch problem.

The database programming language Machiavelli [BO90] is a strongly-typed functional programming language similar to SML [MTHM90] and oriented toward relational and object-oriented databases. SPIDER takes an approach similar to Machiavelli but is oriented toward types for knowledge representation and natural language processing and does not require as many built-in specialized features as Machiavelli. In addition to being strongly-typed and functional, both of these also support polymorphism and have a network model (generalized feature structures and complex objects, respectively).

Machiavelli includes sets, record structures, cyclic structures, relational operations (such as join or project), and classes for object-oriented programming. It supports relational and object-oriented database tasks (querying, views, updating, object creation, etc.) with built-in functions for field selection, field modification, set union, cartesian product, mapping, natural join, projection, and class and instance creation.

SPIDER does not have as many built-in specialized features as Machiavelli, but is oriented toward types for knowledge representation and natural language processing and thus has operations for dealing effectively with sets, feature structures, cyclic structures, unification, and modularization. SPIDER supports knowledge representation tasks by having functions for querying, storage, unification, subsumption (and type) checking, and definition of reasoning methods.

In addition, SPIDER serves as an interface between a functional application language (SML or Common Lisp) and a declarative knowledge representation language (WEB). This allows the semantic binary data model to be manipulated declaratively while remaining compatible with the desire to develop applications in a functional language [PK90]. SPIDER also contains operations for set-like types which reduces impedance mismatch related to set versus element programming [BM88]. This corresponds to multi-valued attributes (or a set-valued range) in the knowledge base.

Another programming language similar to SPIDER is LIFE [AKL88] which integrates functional, logic, and object-oriented programming for application to natural language processing. LIFE is more suited to knowledge representation applications than database ones because LIFE does not have a persistent knowledge store. LIFE supports the natural language processing tasks of syntactic analysis, semantic constraints (such as agreement or selectional restrictions [KF63]), anaphoric resolution, and lexical and grammatical definition. It has built-in operations for function definition (with a pattern-directed syntax), relational rules (like Prolog), unification, and subtyping and type intersection on structured types with coreference (ψ-types [AK84]). Although not all of these operations are necessary for a KBPL, we include all of these except relational rules in SPIDER, because SPIDER is intended to support natural language processing. Relational rules are best developed on top of SPIDER so they can be tailored to the application.

In summary, the desired structural and behavioral features for a KBPL are:

1. The knowledge base structure, which the KBPL accesses, must contain cyclic associations, hierarchical modularization, disjunctive data, and stored abstractions. The knowledge base must support extensional equality (value identity), and a declarative query language.
2. The KBPL must be strongly-typed and extensible. It must support inferencing and unification, be oriented toward knowledge representation tasks, and have a well-specified semantics.

Our knowledge base programming language SPIDER has these features.

Chapter 8
Conclusions

Ah, when to the heart of man
Was it ever less than treason
To go with the drift of things,
To yield with grace to reason,
And bow and accept the end
Of a love or a season?
— Robert Frost

By implementing a knowledge base design tool with a strong theoretical foundation, we have demonstrated that such a tool is possible and useful. Our work has contributed to the general understanding of knowledge bases. We also have discovered advantages and weaknesses in developing this kind of architecture which may be generally useful. We discuss some of these here.

WEAVE has a layered architecture with four levels: a persistent, vivid knowledge store, a persistent graph logic programming language WEB, an extensible knowledge base programming language SPIDER, and a knowledge base manager. The user interacts with WEAVE directly through the knowledge base manager, through a problem solver interface or through a natural language interface.

WEAVE is implemented in Allegro Common Lisp using CLOS on a DecStation 3100. It currently consists of 7000 lines of lisp code and about 4000 lines of programs written in WEB and SPIDER which are used to test and demonstrate WEAVE. The persistent knowledge store, WEB and SPIDER are completely implemented as described in this dissertation. The problem solver interface is implemented as described in chapter 6 but should be tweaked to be more useful. The implementation of the knowledge base manager has only just begun, but all examples given here should work as shown. The natural language interface has not been implemented, but enough previous work with other natural language systems has been done that the queries discussed should work as claimed. Enough implementation was done to validate the difficult theoretical issues, such as the constructive type theory rules needed and the use of sets as generalized multi-valued attributes and inductive types, and the remaining implementation is straightforward.

At times it was difficult to determine where the lines should be drawn between the different layers, but it seems the current architecture works well, and there is a very simple and clean interface between the layers. It was also difficult to decide whether certain features should be included in WEAVE or left as applications to be developed on top of WEAVE. We tried to keep the languages as simple as possible while still allowing for possibly desired knowledge base designs. For example, SPIDER is a restricted language with only the basic functional constructs, but we developed the PRODUCT type constructor to make SPIDER more useful for knowledge base applications. This made the layered architecture slightly more difficult to develop, but the gain in organization and modularity more than outweighed the slight delays from deciding in which module certain features should go. The delays also disappeared as the layers began to take shape. It appears that the layered architecture has some very strong advantages which have not been fully developed by knowledge base or representation systems. It is a very useful approach in separating structure from type (behavioral) information. It also allows layers to be developed which are based on different paradigms when a clean interface occurs between them, and it can be used to isolate functions which interact with users and external systems, which are more likely to change.

WEB is a very expressive graph logic programming language. It appears that binary predicates are very useful for representation, especially when isolated from the end user. The notion of higher order logic is also very useful for designing new representations schemes even if the representation can theoretically be expressed in a first order logic. The graph querying algorithm is an important technical contribution especially when allowing for higher order graph logic. The decision to implement WEB in Common Lisp was purely a practical one, and a Prolog (or LIFE) based implementation might be able to handle a less expressive, but still very useful, form of graph logic with a simpler and possibly more efficient implementation.

Constructive type theory is a very powerful theory which is useful for knowledge base design and representation. It may be useful even without the underlying structural representation in WEB, but constructive type theory seems to be more useful as is used here for dealing with behavior. Very little of constructive type theory's expressive capability was used. This led to a very clean theory and implementation, but it is not clear that such an expressive representation of types was necessary. A more restricted theory may be sufficient. The constructive nature of the theory was useful in formalizing an operational semantics and insuring that knowledge base queries will halt. If this could be done some other way, then any object-oriented front-end (type system with inheritance) might also be useful. The separation of type from structure information is a useful notion, especially in inheritance systems, but that was not made sufficiently clear in this work.

The integration of generalized sets from the knowledge store through the knowledge base manager was a difficult design problem, but led to an extremely clean theory and implementation. In general, the simultaneous development of theories and implementation was extremely useful and many parts of this work could not have been developed without both approaches.

8.1 Contributions

The aim of this dissertation is to make developing knowledge-intensive applications easier by providing a methodology for designing its knowledge base. This is done by setting up a translation from a natural, graphical representation of knowledge to a traditional programming language representation which is one-to-many and reversible. To simplify our task we made two assumptions:

1. The natural representation of domain knowledge contains only symbolic and/or graphical data. We did not deal with video images or acoustical data. Thus, WEB needs only to store symbolic and graphical data.
2. The application is implemented in a functional programming language, such as Lisp or SML [MTHM90].

The primary contributions of this dissertation are:

1. graph querying — a mechanism for retrieving graphs from a persistent knowledge store which match a (partial) specification in graph logic,
2. formalizing graphs as a binary logic that forms the basis of a logic programming language,
3. extensions to constructive type theory and the creation of new type constructors MVA(A) and PRODUCT(A,B) which allow data types to be created that have a structure analogous to graphs,
4. inference rule construction algorithms which make constructive type theory easier to use,

5. an operational semantics for data types created by constructive type theory,
6. the novel integration of theoretical and practical techniques from knowledge representation, natural language semantics, programming languages, and databases, and
7. the application of a knowledge base design process to problem solving, natural language processing, and molecular biology.

Because constructive type theory has been used primarily as a basis for mathematical proofs, we modified it to be applicable to knowledge base design. We developed a generalized notion of set-valued data constructors called *inductive types*. We modified constructive type theory to handle inductive types by introducing set-valued variables to the inference rules which range over subsets of a type and by introducing induction variables which work analogously to recurse variables in recursive types to refer to the computation which remains in obtaining the desired, canonical form. We also developed a type constructor PRODUCT which creates a modified cartesian product of two types which can be used to create binary functions in a manner analogously to unary ones. This allows methods over multiple types to still be associated with one (product) type which lends itself to a much stronger organization of types and methods. It also can help in specifying data model definitions.

To make constructive type theory useful for knowledge base design, we developed algorithms which automatically create all the inference rules for a type constructor when given a type definition in SPIDER. This was possible because of the restrictions which we placed on the type constructors which can be formed. Although these restrictions are very restrictive in terms of the theoretical expressiveness of constructive type theory, they still allow for a wide variety of knowledge base types to be defined. We have developed algorithms for the allowed kinds of type constructors in SPIDER: simple, recursive, inductive, product, and all combinations of them.

These modifications to constructive type theory, the algorithms which automatically construct inference rules, and the formalization of an operational semantics for the inference rules allow for the flexible and powerful definition of types for knowledge base design.

We described our general process for designing knowledge bases. The knowledge base design process is:

1. **Create a graphical sketch.** This should capture the structure and semantics of knowledge for the application.

2. **Abstract common features of the sketch.** These are sections of the graph that can be used to build and manipulate the graph in a meaningful way. They are specified in the graph description language WEB.
3. **Group the abstractions into data types.** These graph abstractions become data constructors for the type.
4. **Implement methods on the type.** These are implemented in the strongly-typed functional programming language SPIDER.
5. **Collect the types and methods to form a data model.** This forms the data model for the application's knowledge base.

We demonstrated our process on the problems of integrating heterogeneous genome maps and constraint-based problem solving. We demonstrated the process on a simple representation for distance information in genome maps and explained how queries can be asked of the knowledge base. We showed how order information can be represented in a similar fashion. Even at this preliminary stage, the results have proven to be useful and extremely promising for solving difficult problems in molecular biology.

8.2 Future Research Directions

There are many different directions in which this work could progress, and some have been mentioned before. We could extend WEAVE from a knowledge base design tool to a knowledge base development tool and enter into work on extensible databases as applied to practical knowledge base systems. Any of the four layers of WEAVE can be extended in many ways. The application to problem solving or natural language can be pursued, and a natural language interface to a molecular biology knowledge base would be extremely useful.

We envision WEAVE as the first step in developing a complete graphical, knowledge base development environment. To use this ideal tool, application developers would sit down at a graphical workstation to draw graphs and diagrams which they (or an expert) would use to solve realistic problems in the domain of interest. This would be an extension of what is currently available in Computer Aided Design. The developer then uses tools to create graphical icons and organizational structures which are needed for problem solving. The goal at this step is to develop a graphical environment which a human problem solver could use to organize the information needed for problem solving in the domain, no matter how complex the information or how tedious the problem solving. The desire is for a flexible, open-ended graphical environment which allows for the creative representation of problem states.

At this step in the process, the application developer has created a graphical domain which is tightly coupled to a natural way of representing the knowledge needed for problem solving. Now, the knowledge base development tool must be used to create a programming environment where the representations can be used efficiently. The desire is to make this transition as painless and straightforward as possible. To do this, we have assumed that the problem solvers are implemented in a traditional (functional) programming language; this could be generalized to other programming language paradigms.

8.3 Conclusion

We have developed a process, theories, and tools for knowledge base design. We used powerful techniques from different areas of computer science wherever possible and developed our own techniques when they were not. We re-investigated older work using new approaches and examined recent advances. We discovered an area others had missed and solved problems others had attempted and failed. We applied our work to existing problems and showed the advantages of our approach. We then found a realistic problem that many people wanted solved and where our work was very applicable. We used our knowledge base design process to find a solution.

Appendix A: SPIDER Syntax

We give the semantics for the SPIDER forms **defstype** and **defsfun**.

The syntax of **defstype** is:

$$\langle \textit{defstype form} \rangle ::= \underline{\texttt{defstype}}\ \langle \textit{name} \rangle\ (\langle \textit{type par} \rangle^{+})\ \langle \textit{scons def} \rangle^{+}$$

$$\langle \textit{scons def} \rangle ::= \langle \textit{scons} \rangle\ \langle \textit{par type} \rangle\ = \ \langle \textit{wcons name} \rangle \langle \textit{con key forms} \rangle^{*}$$

$$\langle \textit{type par} \rangle ::= \langle \textit{variable} \rangle$$

$$\langle \textit{type spec} \rangle ::=$$

$$\langle \textit{par type} \rangle ::= \text{type spec with no variables}$$

$$\langle \textit{con key forms} \rangle ::= \underline{:\mathtt{BASE}-\mathtt{CASE}}\ \langle \textit{scons} \rangle$$

The **defsfun** form is defined by:

```
defsfun <name> <type> <args>
   <pattern> => <expr>
   <pattern> => <expr>
   <pattern> => <expr> ...
```

where the $\langle \textit{pattern} \rangle$'s are sufficient to cover the types (as explained below).

$$
\begin{aligned}
\langle \textit{type} \rangle &::= \langle \textit{SPIDER type} \rangle \ \ | \ \ \underline{(}\ \langle \textit{SPIDER type} \rangle^{n \geq 1}\ \underline{)} \\
\langle \textit{args} \rangle &::= \underline{(}\ \langle \textit{variable} \rangle^{*}\ \underline{)} \\
\langle \textit{pattern} \rangle &::= \langle \textit{constructor expr} \rangle^{n}\ [\ \underline{\texttt{OR}}\ \langle \textit{constructor expr} \rangle^{n}\ ]^{*}\ [\ \langle \textit{where clause} \rangle\ ] \\
&\qquad \text{— } \textit{same } n \text{ as } \langle \textit{type} \rangle \\
\langle \textit{constructor expr} \rangle &::= \langle \textit{constructor} \rangle \\
&\quad\ \ |\ \ \underline{\{}\ \langle \textit{constructor} \rangle\ [\ \underline{::}\ \langle \textit{induction var} \rangle\ ]\ \underline{\}} \\
\langle \textit{where clause} \rangle &::= (\ \underline{\texttt{where}}\ \langle \textit{constraint} \rangle^{+}\ \underline{\Rightarrow}\ \langle \textit{expr} \rangle)^{+}\ \underline{\texttt{otherwise}} \\
\langle \textit{constraint} \rangle &::= \langle \textit{variable} \rangle\ \underline{\texttt{eq}}\ \langle \textit{variable} \rangle \\
&\quad\ \ |\ \ \langle \textit{variable} \rangle\ \underline{\texttt{neq}}\ \langle \textit{variable} \rangle \\
&\quad\ \ |\ \ \langle \textit{variable} \rangle\ \underline{\texttt{in remaining}}\ \langle \textit{variable} \rangle \\
&\quad\ \ |\ \ \langle \textit{variable} \rangle\ \underline{\texttt{notin remaining}}\ \langle \textit{variable} \rangle \\
\langle \textit{induction var} \rangle &::= \langle \textit{variable} \rangle
\end{aligned}
$$

A SPIDER expression is defined by:

$$
\begin{aligned}
\langle expr \rangle ::= \; & \underline{\texttt{LET}} \; [\; \langle variable \rangle \; \underline{\equiv} \; \langle expr \rangle \;]^* \; \underline{\texttt{IN}} \; \langle expr \rangle \\
\mid \; & \underline{\texttt{CASE}} \; \langle variable \rangle \; \underline{\texttt{OF}} \; [\; \langle pattern \rangle \; \underline{\Rightarrow} \; \langle expr \rangle \;]^+ \\
\mid \; & \langle variable \rangle \\
\mid \; & \langle fun\ call \rangle \\
\mid \; & \langle recurse\ form \rangle \\
\langle fun\ call \rangle ::= \; & \langle constant \rangle \; \underline{(} \; \langle expr \rangle^* \; \underline{)} \\
\langle recurse\ form \rangle ::= \; & \underline{(} \; \underline{\texttt{RECURSE}} \underline{)} \\
\mid \; & \underline{\texttt{RECURSE}} \; \underline{(} \; \langle recursive\ intro\ variable \rangle \; \underline{)} \\
\mid \; & \underline{\texttt{RECURSE}} \; \underline{(} \; \langle induction\ var \rangle \; \underline{)} \\
\mid \; & \underline{\texttt{RECURSE}} \; \underline{(} \; \langle recursive\ intro\ variable \rangle \; \langle recursive\ intro\ variable \rangle \; \underline{)} \\
\mid \; & \underline{\texttt{RECURSE}} \; \underline{(} \; \langle induction\ var \rangle \; \langle induction\ var \rangle \; \underline{)}
\end{aligned}
$$

where the variables in a recurse expression are either induction variables or recursive introduction variables (but not both). The form

$$RECURSE(\; \langle recursive\ intro\ variable \rangle \; \langle recursive\ intro\ variable \rangle \;)$$

can only occur in a function on a PRODUCT TYPE, and the two recursive introduction variables must come one from each half of the product.

The semantics of ⟨*defsfun form*⟩ are TransDef⟦⟨*defsfun form*⟩⟧.

Appendix B: Built-in SPIDER Types

B.1 MVA Type

MVA-formation

$$\frac{A\ type}{MVA(A)\ type}$$

$\emptyset$-introduction

$$\frac{}{\emptyset \in MVA(A)}$$

$\cup$-introduction

$$\frac{\begin{array}{l} a \in A \\ r \in MVA(A) \end{array}}{\{a\} \cup r\ \in MVA(A)}$$

MVA-elimination

$$\frac{\begin{array}{l} [\![w \in MVA(A) \quad \triangleright \quad C[w]\ type]\!] \\ x \in MVA(A) \\ b \in C[\emptyset] \\ [\![\ a \in A \\ \quad r \in MVA(A) \\ \quad i \in C[r] \\ \quad \triangleright \quad z(a,r,i)\ \in\ C[\{a\} \cup r] \\ ]\!] \end{array}}{MVA\text{-}elim(x,b,z)\ \in\ C[x]}$$

The computation rules are:

$$MVA\text{-}elim(\emptyset, b, z) \ = \ b \ \in \ C[\emptyset]$$

$$MVA\text{-}elim(\{a\} \cup r, b, z) \ = \ z(a, r, MVA\text{-}elim(r, b, z)) \ \in \ C[\{a\} \cup r]$$

B.2 Product

The product type constructor depends upon the recursive and/or inductive nature of its constituents. Because these are embedded in the introduction rules of the constituents in this presentation of constructive type theory, there is not a clean form for the PRODUCT(A,B) type constructor. Thus, it is omitted.

B.3 Symbol

Symbols are created in the WEB knowledge base when used. Thus, their SPIDER type consists of a theoretically infinite (but countable) collection of distinct elements.

Appendix C: Type Definitions

C.1 Binary Tree

BinTree-formation

$$\frac{A\ type}{BinTree(A)\ type}$$

leaf-introduction

$$\frac{a \in A}{\mathrm{leaf}(a) \in BinTree(A)}$$

node-introduction

$$\frac{l \in BinTree(A) \qquad r \in BinTree(A)}{\mathrm{node}(l, r) \in BinTree(A)}$$

BinTree-elimination

$$\frac{\begin{array}{ll} [\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] & \text{— type premise} \\ x \in BinTree(A) & \text{— major premise} \\ [\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]]\!] & \text{— leaf premise} \\ [\![\ l \in BinTree(A) & \text{— node premise} \\ \quad r \in BinTree(A) & \\ \quad rec\_l \in C[l] & \\ \quad rec\_r \in C[r] & \\ \quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] & \\ ]\!] & \end{array}}{BinTree\text{-}elim(x, leaf\_abs, node\_abs) \ \in \ C[x]}$$

leaf-computation

$$\frac{\begin{array}{l} [\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\ a \in A \\ [\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]]\!] \\ [\![\ l \in BinTree(A) \\ \quad r \in BinTree(A) \\ \quad rec\_l \in C[l] \\ \quad rec\_r \in C[r] \\ \quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] \\ ]\!] \end{array}}{BinTree\text{-}elim(\mathrm{leaf}(a), leaf\_abs, node\_abs) = leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)]}$$

node-computation

$$
\begin{array}{l}
[\![w \in BinTree(A) \quad \triangleright \quad C[w]\ type]\!] \\
l \in BinTree(A) \\
r \in BinTree(A) \\
[\![\ a \in A \quad \triangleright \quad leaf\_abs(a) \ \in \ C[\mathrm{leaf}(a)] \ ]\!] \\
[\![\ l \in BinTree(A) \\
\quad r \in BinTree(A) \\
\quad rec\_l \in C[l] \\
\quad rec\_r \in C[r] \\
\quad \triangleright \quad node\_abs(l, r, rec\_l, rec\_r) \ \in \ C[\mathrm{node}(l, r)] \\
]\!] \\
\hline
BinTree\text{-}elim(\mathrm{node}(l, r), leaf\_abs, node\_abs) \\
\qquad\qquad\quad = \ node\_abs(l, r, BinTree\text{-}elim(l, leaf\_abs, node\_abs), \\
\qquad\qquad\qquad\qquad\qquad BinTree\text{-}elim(r, leaf\_abs, node\_abs)) \\
\quad \in \ C[\mathrm{node}(l, r)]
\end{array}
$$

C.2 Boolean

Boolean-formation

$$\frac{}{\mathit{Boolean\ type}}$$

true-introduction

$$\frac{}{\mathrm{true} \in \mathit{Boolean}}$$

false-introduction

$$\frac{}{\mathrm{false} \in \mathit{Boolean}}$$

Boolean-elimination

$$\frac{\begin{array}{ll} [\![w \in \mathit{Boolean} \quad \triangleright \quad C[w]\ \mathit{type}]\!] & \text{— type premise} \\ x \in \mathit{Boolean} & \text{— major premise} \\ \mathit{true\_val} \in C[\mathrm{true}] & \text{— true-premise} \\ \mathit{false\_val} \in C[\mathrm{false}] & \text{— false-premise} \end{array}}{\mathit{Boolean\text{-}elim}(x, \mathit{true\_val}, \mathit{false\_val}) \ \in \ C[x]}$$

true-computation

$$\frac{\begin{array}{l} [\![w \in \mathit{Boolean} \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\ \mathit{true\_val} \in C[\mathrm{true}] \\ \mathit{false\_val} \in C[\mathrm{false}] \end{array}}{\mathit{Boolean\text{-}elim}(\mathrm{true}, \mathit{true\_val}, \mathit{false\_val}) \ = \ \mathit{true\_val} \ \ \in C[\mathrm{true}]}$$

false-computation

$$\frac{\begin{array}{l} [\![w \in \mathit{Boolean} \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\ \mathit{true\_val} \in C[\mathrm{true}] \\ \mathit{false\_val} \in C[\mathrm{false}] \end{array}}{\mathit{Boolean\text{-}elim}(\mathrm{false}, \mathit{true\_val}, \mathit{false\_val}) \ = \ \mathit{false\_val} \ \ \in C[\mathrm{false}]}$$

C.3 Complex Object

CObj-formation

$$\frac{A\ type}{CObj(A)\ type}$$

cco-introduction

$$\frac{\mathbf{l} \subseteq Labels \qquad \mathbf{v} \subseteq A \qquad n \in Ids}{\mathrm{cco}(\mathbf{l}, \mathbf{v}, n) \ \in \ CObj(A)}$$

CObj-elimination

$$\frac{\begin{array}{l} [\![w \in CObj(A) \quad \triangleright \quad C[w]\ type]\!] \\ x \in CObj(A) \\ [\![\mathrm{id}() = n \ \in \ Ids \quad \triangleright \quad \mathrm{cco}(\emptyset, \emptyset, \mathrm{id}()) = n \ \in \ CObj(A)]\!] \\ b \in C[\mathrm{id}()] \\ [\![\ l \in Labels \\ \quad v \in A \\ \quad \mathbf{rl} \subseteq Labels \\ \quad \mathbf{rv} \subseteq A \\ \quad i \in C[\langle \mathbf{rl}, \mathbf{rv} \rangle] \\ \quad n \in Ids \\ \quad \triangleright \quad z(l, v, \mathbf{rl}, \mathbf{rv}, i, n) \ \in \ C[\mathrm{cco}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \\ ]\!] \end{array}}{CObj\text{-}elim(x, b, z) \ \in \ C[x]}$$

$$CObj\text{-}elim(\mathrm{id}(), b, z) \ = \ b \ \in \ C[\mathrm{id}()]$$

$$\begin{aligned} CObj\text{-}elim(\mathrm{cco}(\{\mathrm{l}\} \cup \mathbf{rl}, \{\mathrm{v}\} \cup \mathbf{rv}, \mathrm{n}), \mathrm{b}, \mathrm{z}) \ &= \ z(l, r, \mathbf{rl}, \mathbf{rv}, CObj\text{-}elim(\mathrm{cco}(\mathbf{rl}, \mathbf{rv}, n), b, z), n) \\ &\in \ C[(\mathrm{cco}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \end{aligned}$$

C.4 Distance Type

Distance-formation

$$\frac{}{\textit{Distance type}}$$

distance-introduction

$$\frac{m1 \in \textit{Distance} \qquad m2 \in \textit{Distance} \qquad e \in \textit{Estimate}}{\textit{distance}(m1, m2, e) \in \textit{Distance}}$$

The elimination and computation rules for DISTANCE (allowing for multiple estimates) are:

Distance-elimination

$$\frac{\begin{array}{l} [\![w \in \textit{Distance} \quad \triangleright \quad C[w] \ \textit{type}]\!] \\ x \in \textit{Distance} \\ b \in C[\text{distance}(m1, m2, \emptyset)] \\ [\![\ m1 \in \textit{Marker(Symbol)} \\ \quad m2 \in \textit{Marker(Symbol)} \\ \quad e \in \textit{Estimate} \\ \quad \mathbf{r} \subseteq \textit{Estimate} \\ \quad i \in C[\mathbf{r}] \\ \quad \triangleright \quad z(m1, m2, e, \mathbf{r}, i) \ \in \ C[\text{distance}(m1, m2, \{e\} \cup \mathbf{r})] \\ ]\!] \end{array}}{\textit{Distance-elim}(x, b, z) \ \in \ C[x]}$$

$$\textit{Distance-elim}(\text{distance}(m1, m2, \emptyset), b, z) \ = \ b \ \in \ C[\text{distance}(m1, m2, \emptyset)]$$

$$\begin{aligned} &\textit{Distance-elim}(\text{distance}(m1, m2, \{e\} \cup \mathbf{r}), b, z) \\ &\quad = \ z(m1, m2, e, \mathbf{r}, \textit{Distance-elim}(\text{distance}(m1, m2, \mathbf{r}), b, z)) \ \in \ C[(\text{ele}(m1, m2, \{e\} \cup \mathbf{r})] \end{aligned}$$

C.5 Feature Structure

FS-formation

$$\frac{A\ \mathit{type}}{FS(A)\ \mathit{type}}$$

cfs-introduction

$$\frac{\begin{array}{l}\mathbf{l} \subseteq \mathit{Labels} \\ \mathbf{v} \subseteq FS(A) \\ n \in \mathit{Ids}\end{array}}{\mathrm{cfs}(\mathbf{l}, \mathbf{v}, n) \ \in \ FS(A)}$$

FS-elimination

$$\frac{\begin{array}{l}[\![w \in FS(A) \quad \triangleright \quad C[w]\ \mathit{type}]\!] \\ x \in FS(A) \\ [\![\mathrm{id}() = n \ \in \ \mathit{Ids} \quad \triangleright \quad \mathrm{cfs}(\emptyset, \emptyset, \mathrm{id}()) = n \ \in \ FS(A)]\!] \\ b \in C[\mathrm{id}()] \\ [\![\ l \in \mathit{Labels} \\ \quad v \in FS(A) \\ \quad \mathbf{rl} \subseteq \mathit{Labels} \\ \quad \mathbf{rv} \subseteq FS(A) \\ \quad i \in C[\langle \mathbf{rl}, \mathbf{rv} \rangle] \\ \quad h \in C[v] \\ \quad n \in \mathit{Ids} \\ \quad \triangleright \quad z(l, v, \mathbf{rl}, \mathbf{rv}, i, h, n) \ \in \ C[\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \\ ]\!]\end{array}}{\mathit{FS\text{-}elim}(x, b, z) \ \in \ C[x]}$$

$$\mathit{FS\text{-}elim}(\mathrm{id}(), b, z) \ = \ b \ \in \ C[\mathrm{id}()]$$

$$\begin{aligned} & \mathit{FS\text{-}elim}(\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n), b, z) \\ & \qquad = \ z(l, r, \mathbf{rl}, \mathbf{rv}, \mathit{FS\text{-}elim}(\mathrm{cfs}(\mathbf{rl}, \mathbf{rv}, n), b, z), \mathit{FS\text{-}elim}(v, b, z), n) \\ & \quad \in \ C[(\mathrm{cfs}(\{l\} \cup \mathbf{rl}, \{v\} \cup \mathbf{rv}, n)] \end{aligned}$$

C.6 List Type

List-formation

$$\frac{A\ type}{List(A)\ type}$$

nil-introduction

$$\frac{}{\text{nil} \in List(A)}$$

cons-introduction

$$\frac{\begin{array}{l} a \in A \\ l \in List(A) \end{array}}{\text{cons}(a,l) \in List(A)}$$

List-elimination

$$\frac{\begin{array}{ll} [\![w \in List(A) \quad \triangleright \quad C[w]\ type]\!] & \text{— type premise} \\ x \in List(A) & \text{— major premise} \\ nil\_val \in C[\text{nil}] & \text{— nil-premise} \\ [\![\ a \in A & \text{— cons-premise} \\ \quad l \in List(A) & \\ \quad rec\_l \in C[l] & \\ \quad \triangleright \quad cons\_abs(a,l,rec\_l) \ \in \ C[\text{cons}(a,l)] & \\ ]\!] & \end{array}}{List\text{-}elim(x, nil\_val, cons\_abs) \ \in \ C[x]}$$

nil-computation

$$\frac{\begin{array}{l} [\![w \in List(A) \quad \triangleright \quad C[w]\ type]\!] \\ nil\_val \in C[\text{nil}] \\ [\![\ a \in A \\ \quad l \in List(A) \\ \quad rec\_l \in C[l] \\ \quad \triangleright \quad cons\_abs(a,l,rec\_l) \ \in \ C[\text{cons}(a,l)] \\ ]\!] \end{array}}{List\text{-}elim(\text{nil}, nil\_val, cons\_abs) \ = \ nil\_val \ \in \ C[\text{nil}]}$$

cons-computation

$$
\begin{array}{l}
[\![w \in \mathit{List}(A) \quad \triangleright \quad C[w] \ \mathit{type}]\!] \\
a \in A \\
l \in \mathit{List}(A) \\
\mathit{nil\_val} \in C[\mathrm{nil}] \\
[\![\ a \in A \\
\quad l \in \mathit{List}(A) \\
\quad \mathit{rec\_l} \in C[l] \\
\quad \triangleright \quad \mathit{cons\_abs}(a, l, \mathit{rec\_l}) \ \in \ C[\mathrm{cons}(a, l)] \\
]\!] \\
\hline
\mathit{List\text{-}elim}(\mathrm{cons}(a, l), \mathit{nil\_val}, \mathit{cons\_abs}) \\
\qquad\qquad = \ \mathit{cons\_abs}(a, l, \mathit{List\text{-}elim}(l, \mathit{nil\_val}, \mathit{cons\_abs})) \\
\quad \in \ C[\mathrm{cons}(a, l)]
\end{array}
$$

C.7 Set

Set-formation

$$\frac{A\ type}{Set(A)\ type}$$

ele-introduction

$$\frac{\begin{array}{l}\mathbf{a} \subseteq A \\ n \in Ids\end{array}}{\mathrm{ele}(\mathbf{a}, n) \ \in \ Set(A)}$$

Set-elimination

$$\frac{\begin{array}{l}[\![w \in Set(A) \quad \triangleright \quad C[w]\ type]\!] \\ x \in Set(A) \\ [\![n \in Ids \quad \triangleright \quad n \in Set]\!] \\ [\![\text{new-id}() = n \ \in \ Ids \quad \triangleright \quad \mathrm{ele}(\emptyset, \text{new-id}()) = n \ \in \ Set]\!] \\ b \in C[\text{new-id}()] \\ [\![\ a \in A \\ \quad \mathbf{r} \subseteq A \\ \quad i \in C[\mathbf{r}] \\ \quad n \in Ids \\ \quad \triangleright \quad z(a, \mathbf{r}, i, n) \ \in \ C[\mathrm{ele}(\{a\} \cup \mathbf{r}, n)] \\ ]\!]\end{array}}{Set\text{-}elim(x, b, z) \ \in \ C[x]}$$

$$Set\text{-}elim(\text{new-id}(), b, z) \ = \ b \ \in \ C[\text{new-id}()]$$

$$Set\text{-}elim(\mathrm{ele}(\{a\} \cup \mathbf{r}, n), b, z) \ = \ z(a, \mathbf{r}, Set\text{-}elim(\mathrm{ele}(\mathbf{r}, n), b, z), n) \ \in \ C[\mathrm{ele}(\{a\} \cup \mathbf{r}, n)]$$

C.8 Table (Problem-Specific)

Table-formation

$$\frac{A \ type}{Table(A) \ type}$$

table-introduction

$$\frac{l \in Chair(A) \qquad r \in Chair(A)}{\mathrm{table}(l, r) \in Table(A)}$$

Table-elimination

$$\frac{\begin{array}{l}\Gamma,\ w \in \mathit{Table}(A) \ \vdash \ C[w] \ \mathit{type} \qquad \Gamma \ \vdash \ x \in \mathit{Table}(A) \\ \Gamma,\ l \in \mathit{Chair}(A),\ r \in \mathit{Chair}(A) \vdash \ \mathit{table\_abs}(l, r) \in C[\mathrm{table}(l, r)]\end{array}}{\Gamma \ \vdash \ \mathit{Table\text{-}elim}(x, \mathit{table\_abs}) \in C[x]}$$

The computation rule for $\mathit{Table}(A)$ is:

$$\mathit{Table\text{-}elim}(\mathrm{table}(l, r), \mathit{table\_abs}) \ = \ \mathit{table\_abs}(l, r) \ \in \ C[\mathrm{table}(l, r)]$$

References

[AB87] Malcolm Atkinson and Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19:105–190, June 1987.

[Abi89] Serge Abiteboul. Towards a deductive object-oriented database language. In *DOOD '89*, 1989. See [KNN89].

[Abr74] J. R. Abrial. Data semantics. In J. W. Klimbie and K. L. Koffeman, editors, *IFIP Working Conference on Data Base Management*, pages 1–59, Amsterdam, April 1974. IFIP, North Holland.

[AFS89] Serge Abiteboul, Patrick C. Fischer, and H.-J. Schek, editors. *Nested relations and complex objects in databases*, volume 361 of *Lecture notes in computer science*. Springer-Verlag, Berlin, 1989.

[AJWRR89] Jürgen Allgayer, R.M. Jansen-Winkeln, Carola Reddig, and N. Reithinger. Biderectional use of knowledge in the multi-modal natural language access system XTRA. In *ijcai-89*, Detroit, 1989.

[AK84] Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. PhD thesis, Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1984.

[AKL88] Hassan Aït-Kaci and Patrick Lincoln. Life: A natural language for natural language. Technical Report ACA-ST-074-88, MCC, February 1988.

[AKN86] H. Aït-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):187–215, 1986.

[All90] Jürgen Allgayer. SB-ONE$^+$ — dealing with sets efficiently. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 13–18, 1990.

[All91] James F. Allen. The RHET system. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[And85] John R. Anderson. *Cognitive Psychology and Its Implications*. W.H. Freeman and Co., 2nd edition, 1985.

[ARS90] Jürgen Allgayer and Carola Reddig-Siekmann. What KL-ONE lookalikes need to cope with natural language. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *LNAI*, pages 240–285. Springer-Verlag, 1990.

[Bac86a] Roland C. Backhouse. Notes on Martin-Löf's theory of types, parts 1 and 2. *FACS FACTS*, 1986.

[Bac86b] Roland C. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. Computer Science Notes CS 8606, Dept of Mathematics and Computer Science, University of Groningen, 1986.

[BB90] Francois Bançilhon and Peter Buneman, editors. *Advances in database programming languages*. ACM Press, New York, N.Y., 1990.

[BBG+88] D. S. Batory, J. R. Barnett, J. F. Garza, et al. Genesis: An extensible database management system. *IEEE Trans on Software Engineering*, 14(11), November 1988.

[BBMR89] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori A. Resnick. Classic: a structural data model for objects. *ACM SIGMOD Record*, 18(2):58–67, 1989.

[BC85] Joseph Bates and Robert Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[BCM88] Roland Backhouse, Paul Chisholm, and Grant Malcolm. Do-it-yourself type theory (parts 1 and 2). In *EATCS*, January 1988.

[BE90a] Jon Barwise and John Etchemendy. Information, infons, and inference. In Robin Cooper, Kuniaki Mukai, and John Perry, editors, *Situation Theory and its Applications (volume 1)*, number 22 in Lecture Notes, chapter 2. CSLI, 1990.

[BE90b] Jon Barwise and John Etchemendy. Visual information and valid reasoning. In W. Zimmerman, editor, *Visualization in Mathematics*. Mathematical Association of America, Washington, DC, 1990.

[BFL83a] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, October 1983. A slightly extended version appears in [BL85].

[BFL83b] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. KRYPTON: Integrating terminology and assertion. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 31–35. American Association for Artificial Intelligence, August 1983.

[BGL85] Ronald J. Brachman, Victoria Pigman Gilbert, and Hector J. Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON. In *ijcai-85*, pages 532–539, August 1985.

[BH91] Franz Baader and Bernhard Hollunder. KRIS: knowledge representation and inference system. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[BHS90] Karl Hans Bläsius, Ulrich Hedstück, and J.H. Siekmann. Structure and control of the l-LILOG inference system. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *LNAI*, pages 165–182. Springer-Verlag, 1990.

[BK86] François Bancilhon and S. N. Khoshafian. A calculus of complex objects. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 53–59, 1986.

[BL84] R.J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. AAAI*, pages 34–37, August 1984.

[BL85] R.J. Brachman and H.J. Levesque. *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos, CA, 1985.

[BL87] Lubomir Bic and Craig Lee. A data-driven model for a subset of logic programming. *ACM Transactions on Programming Languages and Systems*, 9(4):618–645, October 1987.

[BM88] François Bancilhon and David Maier. Multi-language object-oriented systems: New answers to old database problems. In Kazuhiro Fuchi and Laurent Kott, editors, *Programming of future generation computers II*. North-Holland, Amsterdam, 1988.

[BMPS+90] Ronald Brachman, Deborah McGuinness, Peter Patel-Schneider, Lori Alperin Resnick, and Alex Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the representation of knowledge*. Morgan-Kaufmann, San Mateo, CA, 1990.

[BO90] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. Dept of Computer and Information Science MS-CIS-90-64, Univ of Pennsylvania, September 1990. (also TR Logic & Computation 24).

[Bra79] R.J. Brachman. On the epistemological status of semantic networks. In Nicholas V. Findler, editor, *Associative Networks - The Representation and Use of Knowledge by Computers*. Academic Press, New York, 1979. Also BBN Report 3807, April 1978.

[Bra80] R.J. Brachman. An introduction to kl-one. In Brachman R. J., editor, *Research in Natural Language Understanding*, pages 13–46. Bolt, Beranek and Newman Inc., Cambridge, MA, 1980.

[Bro84] Michael Brodie. On data models. In Michael Brodie, John Mylopoulos, and Joachim Schmidt, editors, *On conceptual modelling: Perspectives from artificial intelligence, databases, and programming languages*, pages 19–47. Springer-Verlag, New York, 1984.

[BS82] R.J. Brachman and Jim Schmolze. Second kl-one workshop. *AI Magazine*, 3(1):15, winter 1981/1982.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, pages 171–216, August 1985.

[BV91] Samuel Bayer and Marc Vilian. The relation-based knowledge representation of King-Kong. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car92] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, 1992.

[CBP+90] DR Cox, M Burmeister, ER Price, S Kim, and RM Myers. Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes. *Science*, 250:245–250, 1990.

[CCM92] Mariano Consens, Isabel Cruz, and Alberto Mendelzon. Visualizing queries and querying visualizations. *ACM SIGMOD Record*, 21(1):39–46, March 1992.

[CDF+86] M. Carey, D. DeWitt, D. Frank, G. Graefe, et al. The architecture of the EXODUS extensible database system. In *Proc of the International Workshop on Object-Oriented Database Systems*, pages 52–65, New York, 1986. IEEE. Pacific Grove, CA.

[CDG+90] Michael J. Carey, David J. DeWitt, Goetz Graefe, et al. The EXODUS extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in object-oriented database systems*, pages 474–499. Morgan Kaufmann, 1990.

[CK91] James M. Crawford and Benjamin J. Kuipers. Algernon — a tractable system for knowledge-representation. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990.

[Cra89] James M. Crawford. Towards a theory of access-limited logic for knowledge representation. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, May 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, December 1985.

[Dal90] Fred H. Dale. The Rock City boosters. In *The Dell Crossword Puzzle Travel Companion*, page 25. Dell, New York, NY, 1990.

[Dav91] Randall Davis. A tale of two knowledge servers. *AI Magazine*, pages 118–120, Fall Davis91.

[DF84] Edsger Wybe Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren.* Academic Service Den Haag, 1984.

[DK79] Amaryllis Deliyanni and R. Kowalski. Logic and semantic networks. *Communications of the ACM*, 22(3):184–192, March 1979.

[DKM91] C Delobel, M Kifer, and Y Mas, editors. *Deductive and object-oriented databases : Second International Conference, DOOD '91*, volume 566 of *Lecture notes in computer scienc.* Springer-Verlag, Berlin, December 1991.

[EBBK89] David Etherington, Alex Borgida, Ronald Brachman, and Henry Kautz. Vivid knowledge and tractable reasoning: Preliminary report. In *Proc. IJCAI-89*, pages 1146–1152. IJCAI, 1989.

[EO86] Jürgen Edelmann and Bernd Owsnicki. Data models in knowledge representation systems: A case study. In *GWAI-86 und 2.*, pages 69–74. Springer Verlag, 1986.

[Fah79] S.E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press, Cambridge, MA, 1979. Based on Phd thesis, MIT, Cambridge, MA, 1979.

[FH77] R.E. Fikes and G.G. Hendrix. A network-based knowledge representation and its natural deduction system. In *Proc. IJCAI-77*, pages 235–246. IJCAI, 1977.

[Fre79] G. Frege. Begriffsschrift, a formula language modelled upon that of arithmetic, for pure thought. In J. van Heijenoort, editor, *From Frege to Godel: A Source Book In Mathematical Logic, 1879-1931*, pages 1–82. Harvard University Press, Cambridge, MA, 1879.

[Gai91] Brian R. Gaines. Empirical investigation of knowledge representation servers: design issues and applications experience with KRS. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[GM78] H. Gallaire and J. Minker, editors. *Logic and Databases.* Plenum Press, New York, 1978.

[GP92] K Gardiner and D Patterson. The role of somatic cell hybrids in physical mapping. *Cytogenet Cell Genet*, 59:82–85, 1992.

[GPG90] Marc Gyssens, Jan Paradaens, and Dirk Van Gucht. A graph-oriented object model for database end-user interfaces. In *Proceedings of 1990 ACM SIGMOD Conference on Management of Data*, 1990.

[Gua91] Nicola Guarino. A concise presentation of ITL. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[Heu89] Andreas Heuer. A data model for complex objects based on a semantic database model and nested relations. In Serge Abiteboul, Patrick C. Fischer, and Hans-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 297–312. Springer-Verlag, 1989.

[HK87] Richard Hull and Roger King. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys*, 19:201–260, September 1987.

[Hul87] Richard Hull. A survey of theoretical research on typed complex database objects. In Jan Paredaens, editor, *Databases*, chapter 5, pages 193–256. Academic Press, 1987.

[Jon87] Simon L. Peyton Jones. *The implementation of functional programming languages.* Prentice/Hill International, Englewood Cliffs, NJ, 1987.

[Jr.57] Clarence Raymond Wylie Jr. *101 Puzzles in Thought and Logic.* Dover, New York, 1957.

[KBR86] Thomas S. Kaczmarek, Raymond Bates, and Gabriel Robins. Recent developments in NIKL. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986.

[KC86] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1986. Also in [ZM90].

[KF63] Jerrold Katz and Jerry Fodor. The structure of a semantic theory. *Language*, 39:170–210, 1963. Reprinted in Fodor and Katz, eds, *The structure of language*. Prentice-Hall, 1964.

[KMS83] Thomas S. Kaczmarek, W. Mark, and N. Sondheimer. The Consul/CUE Interface: An integrated interactive environment. In *Proc of CHI '83 Human Factors in Computing Systems*, pages 98–102. ACM, December 1983.

[Kni89] Kevin Knight. Unification: a multidisciplinary survey. *ACM Surveys*, 21(1), 1989.

[KNN89] Won Kim, Jean Marie Nicolas, and Shojiro Nishio, editors. *Deductive and object-oriented databases : proceedings of the First International Conference on Deductive and Object- Oriented Databases (DOOD '89).* North Holland, New York, December 1989.

[Kob91] Alfred Kobsa. First experiences with the SB-ONE knowledge representation workbench in natural-language applications. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[Kow79] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[KR86] R. T. Kasper and W. C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics*, pages 235–242, 1986.

[LG90] Douglas B. Lenat and Ramanathan V. Guha. *Building large knowledge-based systems : representation and inference in the Cyc project.* Addison-Wesley Pub., Reading, Mass., 1990.

[Mac88] Robert MacGregor. A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 403–408, Saint Paul, Minnesota, August 1988.

[Mar83] Leo Mark. What is the binary relationship approach? In *Entity-Relationship Approach to Software Engineering.* North-Holland, 1983.

[Mar86] Fred Maryanski. The data model compiler: A tool for generating object-oriented database systems. In *Proc of the International Workshop on Object-Oriented Database Systems*, pages 73–84, New York, 1986. IEEE. Pacific Grove, CA.

[MB87] Robert MacGregor and Raymond Bates. The LOOM knowledge representation language. Technical Report ISI/RS-87-188, USC/Information Sciences Institute, 1987.

[MB89] John Mylopoulos and Michael Brodie. *Readings in artificial intelligence and databases.* Morgan Kaufmann, 1989.

[MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[MDW91] Eric Mays, Robert Dionne, and Robert Weida. K-Rep system overview. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[MH85] Fred Maryanski and S. Hong. A tool for generating semantic database applications. In *COMPSAC 85*, pages 368–375. IEEE, October 1985.

[Min75] Marvin Minsky. A framework for representing knowledge. In Patrick Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, NY, 1975.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy*, pages 153–175, Amsterdam, 1982. North-Holland.

[MTHM90] Robin Milner, Mads Tofte, Robert Harper, and Prateek Misbra. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[NBF91] Robert Nado, Jeffrey Van Baalen, and Richard Fikes. JOSIE: An intergration of specialized representation and reasoning tools. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[Neb88] Bernhard Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, April 1988.

[NS90] Bernhard Nebel and Gert Smolka. Representation and reasoning with attributive descriptions. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *LNAI*, pages 112–139. Springer-Verlag, 1990.

[NvL87] Bernhard Nebel and Kai von Luck. Issues of integration and balancing in hybrid knowledge representation systems. In K. Morik, editor, *German Workshop on Artificial Intelligence 1987*. Springer Verlag, 1987.

[NvL88] Bernhard Nebel and Kai von Luck. Hybrid reasoning in BACK. In Zbignew W. Ras and Lorenza Saitta, editors, *Methodologies for Intelligent Systems*, volume 3, pages 260–269. North-Holland, New York, 1988.

[Oho88] Atsushi Ohori. Semantics of types for database objects. In *2nd International Conference on Database Theory*, volume 326 of *LNCS*. Springer-Verlag, 1988.

[OK89] Bernd Owsnicki-Klewe. Configuration as a consistency maintenance task. In G. Hoeppner, editor, *GWAI-88*. Springer-Verlag, 1989.

[Ott91] Jurg Ott. *Analysis of human genetic linkage*. Johns Hopkins University Press, Baltimore, revised edition, 1991.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[Pig84a] Victoria Pigman. The interaction between assertional and terminological knowledge in KRYPTON. In *Proceedings IEEE Workshop on Principles of*

Knowledge-Based Systems, pages 3–10. IEEE Computer Society, December 1984.

[Pig84b] Victoria Pigman. KRYPTON: Description of an implementation, volume 1. AI Technical Report 40, Schlumberger Palo Alto Research, November 1984.

[PK90] Alexandra Poulovassilis and Peter King. Extending the functional data model to computational completeness. In Francois Bançilhon, Consantion Thanos, and Dennis Tsichritzis, editors, *Advances in Database Technology — EDBT'90*, volume 416 of *Lecture notes in computer science*, pages 75–91. Springer-Verlag, Berlin, 1990.

[Ple91] Udo Pletat. Reasoning over modularized knowledge. In *Notes from 1991 AAAI Fall Symposium on Principles of Hybrid Reasoning*, 1991.

[PM88] Joan Peckham and Fred Maryanski. Semantic data models. *ACM Computing Surveys*, 20:153–189, September 1988.

[PPT91] J. Paredaens, P. Peelman, and L. Tanca. G-log : a declarative graphical query language. In *Deductive and object-oriented databases : Second International Conference, DOOD '91*, volume 566 of *LNCS*, pages 108–128, Berlin, 1991. Springer-Verlag.

[PS84] Peter F. Patel-Schneider. Small can be beautiful in knowledge representation. In *Proc. of the IEEE Workshop on Principles of Knowledge-Based Systems*, pages 11–16, Denver, CO, December 1984. IEEE Computer Society. A revised and extended version is available as AI-TR-37, Schlumberger Palo Alto Research, Oct 1984.

[PS89] Peter Patel-Schneider. Undecidability of subsumption in NIKL. *Artificial Intelligence Journal*, 39:263–272, 1989.

[PSMB+91] Peter Patel-Schneider, Deborah McGuinness, Ronald Brachman, Lori Alperin Resnick, and Alex Borgida. The CLASSIC knowledge representation system: Guiding principles and implementation rationale. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[PT91] Jeff Pan and Jay Tenenbaum. An intelligent agent framework for enterprise integration. *IEEE Trans on Systems, Man, and Cybernetics*, 21(6):1391–1408, November 1991.

[PvL90] Udo Pletat and Kai von Luck. Knowledge representation in LILOG. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *LNAI*, pages 140–164. Springer-Verlag, 1990.

[Qui68] M.R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*. The MIT Press, Cambridge, MA, 1968. Also PhD Thesis, Carnegie Institute of Technology, 1967.

[Ric82] Charles Rich. Knowledge representation languages and predicate calculus: How to have your cake and eat it too. In *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[Ric85] Charles Rich. The layered architecture of a system for reasoning about programs. In *ijcai-85*, pages 540–546, August 1985.

[Ris85] N. Rishe. Semantic modelling of data using binary schemata. Technical Report TRCS85-06, Univ of California, Santa Barbara, 1985.

[Ris86] N. Rishe. On representation of medical knowledge by a binary data model. In X. J. R. Avula G. Leitman, Jr. C. D. Mote, and E. Y. Rodin, editors, *Proc of the 5th International Conference on Mathematic Modelling*, Elmsford, NY, 1986. Pergamon Press.

[Rob73] Don Roberts. *The Existential Graphs of Charles S. Peirce.* Mouton, The Hague, 1973.

[Rou91] Bill Rounds. Situation-theoretic aspects of databases. In Jon Barwise, Jean Mark Gawron, Gordon Plotkin, and Syun Tutiya, editors, *Situation Theory and Its Applications*, chapter 11. Stanford, 1991.

[SB86] M. Stefik and D. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, VI(4):40–62, Winter 1986.

[Sch72] R.C. Schank. Conceptual dependency: A theory of natural language understanding. *Cognitive Psychology*, 3(4):552–631, 1972.

[Sch89a] Manfred Schmidt Schauss. Subsumption in KL-ONE is undecidable. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, May 1989.

[Sch89b] James G. Schmolze. Terminological knowledge representation systems supporting n-ary terms. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, May 1989.

[SG91] Narinder Singh and Michael Genesereth. Epikit: A library of subroutines supporting declarative representations and reasoning. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[SGC79] L.K. Schubert, R.G. Goebel, and N.J. Cercone. The structure and organization of a semantic net for comprehension and inference. In Findler, editor, *Associative Networks - The representation and use of knowledge in computers*, pages 121–175. Academic Press, New York, 1979.

[Shi86] S. Shieber. *An Introduction To Unification-Based Approaches To Grammar.* CSLI, Stanford, CA, 1986.

[Shr91] Howard E. Shrobe. Providing paradigm orientation without implementational handcuffs. *ACM SIGART Bulletin*, 2(3), June 1991. Special issue on Implemented Knowledge Representation and Reasoning Systems.

[Sow84] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA, 1984.

[SPT87] Lenhart K. Schubert, Mary Angela Papalaskaris, and Jay Taugher. Accelerating deductive inference: Special methods for taxonomies, colours and times. In

Nick Cercone and Gordon McCalla, editors, *The Knowledge frontier : essays in the representation of knowledge*, chapter 9. Springer-Verlag, 1987.

[SS91] Yuh-Ming Shyy and Stanley Y.W. Su. K: A high-level knowledge base programming language for advanced database applications. In James Clifford and Roger King, editors, *Proceedings 1991 SIGMOD*, pages 338–347, Denver, CO, May 1991. ACM. Also in SIGMOD Record 20(2) June, 1991.

[THW+88] Rudolph E Tanzi, JL Haines, PC Watkins, GD Stewart, MR Wallace, R Hallewell, C Wong, NS Wexler, PM Conneally, and JF Gusella. Genetic linkage map of human chromosome 21. *Genomics*, 3:129–136, 1988.

[TWS+92] Rudolph E Tanzi, PC Watkins, GD Stewart, NS Wexler, JF Gusella, and JL Haines. A genetic linkage map of human chromosome 21: Analysis of recombination as a function of sex and age. *American Journal Human Genetics*, pages 551–558, 1992.

[VB82] G.M.A. Verheijen and J. Van Bekkum. NIAM: An information analysis method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*. North-Holland, 1982.

[Vil85] Marc Vilain. The restricted language architecture of a hybrid representation system. In *ijcai-85*, pages 547–551, August 1985.

[vLPNS87] Kai von Luck, Christof Peltason, Bernhard Nebel, and Albrecht Schmiedel. The anatomy of the BACK system. KIT-Report 41, Fachbereich Informatik, Technische Universität Berlin, January 1987.

[VM83] Marc Vilain and David A. McAllester. Assertions in NIKL. Technical Report 5421, BBN Laboratories, 1983.

[Vos91] Gottfried Vossen. Bibliography on object-oriented database management. *ACM SIGMOD Record*, 20(1):24–46, March 1991.

[Win70] P.H. Winston. Learning structural descriptions from examples. Technical Report MIT AI-TR-231, MIT, Cambridge, Mass., September 1970.

[WSL+89] Andrew C Warren, SA Slaugenhaupt, JG Lewis, A Chakravarti, and SE Antonarakis. A genetic linkage map of 17 markers on human chromosome 21. *Genomics*, 4:579–591, 1989.

[Zan90] Carlo Zaniolo. Deductive databases — theory and practice. In *Advances in database technology — EDBT '90*, volume 416 of *LNCS*, pages 1–15. Springer-Verlag, 1990.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in object-oriented database systems*. Morgan Kaufmann, San Mateo, CA, 1990.