To appear in the Proceedings of the Fourth International Conference on Information and Knowledge Management. November, 1995. ACM Press.

Application of Knowledge Base Design Techniques to Genetic Markers

Mark Graves

Department of Cell Biology, Baylor College of Medicine
One Baylor Plaza, Houston, TX 77030
email: mgraves@bcm.tmc.edu; *phone:* (713) 798-8271; *fax:* (713) 798-3759

Abstract

Knowledge bases can be quickly accepted into realistic settings if they are developed rapidly and they support critical tasks. Genetics is one area where knowledge management is necessary to provide intelligent access to complex information. Multiple reasoning systems accessing the same knowledge base can contribute directly to the progress of science. Genetics is a rapidly changing field and the rapid development of intelligent systems is becoming crucial for the analysis of information. Genetic data is also being generated at an exponentially growing rate and traditional techniques are being overwhelmed. A flexible knowledge base which stores the information and makes it available to a variety of reasoning systems will speed up the transition of laboratory research to medical practice.

We have developed a process for designing data models for application-specific knowledge bases. This process is supported by a knowledge base design tool, which we call WEAVE, that creates a prototype knowledge base when given a data model definition by a knowledge base developer. The process is formalized using theoretical results from knowledge representation and programming language theory. This mix of theoretical results and their integration in an implemented system has proven useful for a real-world application. We show how our knowledge base design process can be used to develop a data model for *genetic markers*, which are biological constructs used to search for genes associated with genetic diseases.

1. Introduction

Computer science has contributed greatly to many areas of everyday life. The rapid growth of technology and its integration into common culture has been amazing. Molecular genetics is a field which is growing even more rapidly and has a direct influence on the quality of life for everyone. Recent discoveries include the role of genetic factors in cancer, heart disease, and other major internal causes of death (Schiver et al. 1995). Current technology is sufficient for discovering every human gene in the next 10 years, and much faster technologies are under development. A great deal of effort is being spent on developing gene therapies so genetic diseases can be treated or cured. Computer scientists can contribute by providing tools to help geneticists capture and analyze the data.

Intelligent information processing is becoming essential. The exponential growth in the amount of information available to geneticists overwhelms traditional approaches to organizing data in laboratory notebooks (Cinkosky 1995). Geneticists need tools now which will help them capture the data, and soon they will also need tools to help them analyze the data. Intelligent systems become crucial, not because geneticists do not know how to analyze the data manually, but because they need systems which can repeat the analysis process millions of times. At the foundation of large-scale intelligent information processing is knowledge management.

The purpose of this paper is two-fold. One purpose is to describe a system that was developed to build domain-specific knowledge bases (Graves 1993). Although developed without genetics in mind, it has proved to be useful for representing genetics and some parts are being used on a daily basis as support for laboratory research. The other purpose of this paper is to present some aspects of genetics to the computer science community, encouraging computer scientists to apply their knowledge to a field where it can be immediately used to increase longevity and improve quality of life.

2. Knowledge Base Design

Rapid development is crucial for knowledge bases to be useful in genetics. Many laboratory techniques used in genetics today, such as polymerase chain reaction (PCR), were only discovered recently. Revolutionary advances which were discovered 15 years ago, such as Restriction Fragment Length Polymorphism (RFLP) mapping, are now obsolete. Knowledge bases that support reasoning, knowledge mining and other intelligent access mechanisms must be built to support the most current laboratory techniques, or they will be considered obsolete before the development of them is completed.

Although many data capture problems in genetics can be solved with traditional database technology, some of them cannot (Lawrence 1995). Many of the more complex data capture tasks can be supported with systems that incorporate knowledge representation formalisms into a database framework (Graves 1995), but the complexity of genetic data may be difficult to analyze intelligently without sophisticated knowledge management. A first step in the development of knowledge bases to support intelligent information processing is to provide a framework where data can be captured to support current needs and organized in a structure which supports the intelligent analysis to be performed in the future.

To assist in the rapid development of knowledge bases, we have developed a process of knowledge base design which is supported by theoretical results and computational tools. These

knowledge base design techniques have been useful in the design of a knowledge base which stores information about genetic markers.

2.1 Hybrid Architecture

An important aspect of any intelligent system is knowledge representation. When the system is to reason on a large amount of data, two representations are crucial: the representation of the reasoning system and the representation of the data in the system's stored format. With a large system, multiple reasoners will access the data, and they will each work more effectively and efficiently if they can manipulate the data using their own representation scheme. Small systems may duplicate the data to meet the requirements of the reasoning tools, but large systems must store the data in one format. Often, it is useful to store the results of the reasoning back into the underlying knowledge base.

Graphs are a basic computer science construct that geneticists find natural for describing data. They are a good representation for genetic data, because the interconnection of genetic constructs has a graph-like structure. For example, a gene is not only a sequence of four nucleotide bases (abbreviated A,G,C,T) but also is a complex system of introns, exons, promoters, repressors, and other regulatory regions which has a graph-like structure. Proteins are encoded by genes which affect the entire biochemistry of the cell, including (recursively) the transcription of genes.

Graphs are also the foundation of a useful genetics database system. Graph database management systems have as their foundation a data model based on the mathematical definition of a graph as a collection of vertices and edges, which distinguishes them from network and semantic data models. There are several extensions to the definition of graphs which form foundations for different graph data models. One approach to developing a genetics database system would be to create extensions to the graph data model which would support the specific requirements of genetics (Graves 1995a), but the approach we take here is to extend the graph to support the creation of domain-specific knowledge bases.

A problem arises when providing multiple reasoning tools on a knowledge base. A system with multiple reasoning tools must support a variety of representations. Each tool has its own set of requirements, and representations which support one tool may bias the data in a way which will make it less usable by a different tool. This problem may be alleviated by considering each tool's representation language as a view on the representation language of the underlying knowledge base. The representation languages of the tools may be described as a data model.

A data model is a useful mechanism in which to describe the interface to the storage mechanism of a knowledge base. Data models typically describe the representation and manipulation capabilities of a database management system, but they are also useful as an extension to languages which typically describe data in terms of data types. The use of data types to define an interface has been successful in extensible databases, such as EXODUS (Carey 1986), and programming languages such as Modula/2 and SML. We expand on the use of abstract data types by using type constructors as the foundation of user-defined data models.

Our architecture is to have reasoning tools access domain knowledge represented as graphs. The knowledge base access of the tool takes place via an interface, which can be defined by a data model whose operations access the knowledge base. The data model is defined as a collection of type constructors which specify the data which can be stored in the knowledge base. Type constructors are defined by the data model developer within a programming language we call SPIDER. SPIDER is a strongly-typed, functional programming language whose type system is based on constructive type theory (Martin-Lof 1982). SPIDER contains algorithms which create the natural-deduction style inference rules of constructive type theory for a user-defined type constructor. When the user writes programs on the type constructors they are evaluated by a deterministic proof procedure.

Type constructors are defined as a collection of data constructors which define the relationships that can be created and retrieved from the knowledge base. Constructive type theory defines type constructors and data constructors in terms of inference rules, and type constructors create abstract data types. For example, *List(α)*, where α is a type variable, is a type constructor which creates lists; *List(Symbol)* is an abstract data type for list of symbols whose elements are created by the data constructors "pair" and "nil".

Data constructors simplify the underlying knowledge base by isolating the type information to the application. Type information is not represented in the graph but is specific to each data model: this simplifies the maintenance of the knowledge base, especially in a rapidly changing domain. Each data constructor is associated with a graph constructor which defines the structure of information within the knowledge base, while the operations on the data type define the behavior.

Graph constructors are programs that define the vertices and edges in graph. By using vertices created by one constructor as an argument to another, large graphs can be constructed. Because it is useful to use the same programs to both build and query the knowledge base, we have developed a declarative programming language to define graphs, which we call WEB. WEB is based on attribute value formalisms (Nebel & Smolka 1990). One advantage to using graph constructors instead of creating vertices and edges individually is that constraints on the graph may be enforced.

2.2 Process

Developing knowledge bases can be a complex process. To simplify the task, we have developed a knowledge base design process which incorporates features of database design and software engineering. Our process for developing a knowledge base using the tool WEAVE is:

1. **Create a graphical representation of the domain information.** The graphical description should capture the structure and semantics of the knowledge for the application. This step in the process is similar to knowledge elucidation in expert system design, domain analysis in software engineering (Prieto-Diaz & Arango 1991), or conceptual modeling in database design (Brodie 1984).
2. **Isolate subgraphs which are meaningful for specific application tasks.** These abstractions (graph constructors) are sections of the graph that can be used to build and manipulate the graph in a meaningful way. This step is similar to analysis in object-oriented software engineering (Coleman 1994). The subgraphs will be used to define programs in the graph logic programming language WEB.
3. **Define subgraphs as graph constructors, written in WEB.** These graph programs define the subgraphs used to define and query the knowledge base. They become data constructors for the type constructors which create application-specific abstract data types. This step involves writing simple logic programs.

4. **Associate graph constructors with data constructors, written in SPIDER.** These are implemented in the strongly-typed, functional programming language SPIDER. This step requires adding type information to the graph constructors.
5. **Group the data constructors into type constructors.** Data constructors could be used individually, but they simplify the design when they are grouped together into type constructors. These type constructors defined by the knowledge base developer are used in the knowledge base application programs to create abstract data types. Abstract data types isolate the structural constraints of the knowledge from the behavior required. Data constructors should be grouped when they are likely to share functionality in the application, such as creating graphs of similar purpose or requiring similar operations be defined.
6. **Implement operators for the type constructors.** These operators manipulate the values in the type. Because of the foundation of SPIDER in constructive type theory, the operators are defined in terms of the data constructor used to create that value. This step involves writing simple programs in SPIDER.
7. **Collect the type constructors and operators to form a data model.** These type constructors and operators form the data model for the knowledge base.

2.3 Theories

Knowledge base design can deteriorate into *ad hoc* development unless there are theories supporting it which are both powerful enough to express the needed information and natural enough to guide the development in a productive manner. One way of representing structural information is a flexible, graph-theoretic framework. Behavioral information can be stored in a strongly-typed system (as is done in the functional and object-oriented programming language paradigms). Static and dynamic domain information of knowledge-intensive applications may be organized in terms of a data model.

We import attribute value description languages from knowledge representation (Brachman & Schmolze 1985; Ait-Kaci 1984) and natural language semantics (Kasper & Rounds 1986; Sowa 1984) to represent the structure of knowledge in a graph framework, and we import constructive type theory (Martin-Lof 1982) from programming language research to represent type information in a manner that lends itself to the automatic generation of inference rules on the type. These, together with a persistent, binary (dyadic) logic knowledge store (Deliyanni & Kowalski 1979; Bic & Lee 87) and an algebraic approach to data models, form a strong, theoretical foundation for knowledge base design upon which we have implemented the tool WEAVE.

Attribute value description languages (Nebel & Smolka 1990) represent information in a framework of concepts and attributes which has a Tarski-style, model-theoretic semantics but can represent the information usually found in graphical notations. These have been used for a variety of applications, and we have developed an attribute value description language, WEB, which is geared toward knowledge base design. Binary predicates are used as the foundation of our knowledge base because they form a simple basis which can be uniformly implemented and treated as attributes in semantic nets, roles in frames, arcs in graphs, etc. This is important for extensibility. They form the foundation for our graph logic which is implemented as the graph programming language WEB.

Constructive (intuitionistic) mathematics is a non-classical approach which does not allow for indirect proofs. Constructive type theory 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. Because constructive type theory has been used primarily as a basis for mathematical proofs (Constable et al. 1986; Paulson 1989), it must be modified 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. The built-in type constructor in SPIDER for generalized multi-valued attributes can be used to define set-like types. 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.

Data models are organized using an algebraic approach and are built up using the constructive type definitions and the operations which are developed as methods on the types. Integrity constraints are not included in the data model, though they are supported by constructive type theory. So far, combining strong-typing with the flexibility of multiple data types that access the same graph has been sufficient to provide appropriate integrity constraints.

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

2.4 Tools

Our architecture for the knowledge base design tool is developed in four layers: the physical (lowest) level, the structural level, the data type level, and the data model level. The physical level uses a binary logic knowledge store to organize the data and its abstractions. The structural level uses the graph logic we have developed to define the structure of the knowledge in WEB. The data type level uses constructive type theory to define the data types for the application in SPIDER. The fourth level uses an algebraic approach to define the data models. We describe only the middle two layers in this paper. All four levels are combined into the tool WEAVE as outlined in Figure 1, where the solid figures describe aspects of WEAVE, the dashed figures denote what would be added outside of WEAVE to use it, and the arrows describe the information flow of the system.

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, but less flexible, implementation which still has the functional and interface specification of the original, theory-guided design. It also appears that in many cases some of these efficient implementations may be automatically compiled from the original theoretical definitions.

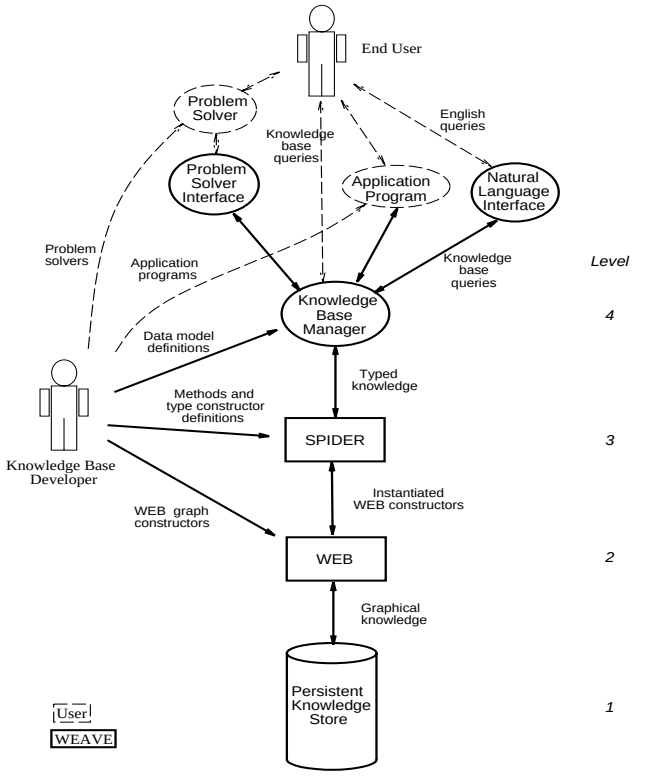


Figure 1: WEAVE System Architecture

We have not found it productive to take these steps, in part because biologists are much more interested in reducing manual day-long processes down to a few minutes than they are in speeding up unattended processing.

WEAVE is implemented in 7,000 lines of Allegro Common Lisp with CLOS (Graves 1993). Since WEAVE was developed, partial re-implementations of the lower two levels have been developed. These implementations were developed to help support laboratory applications with different data storage requirements. The physical layer has been implemented is Smalltalk with persistent storage in either the Smalltalk image, Unix files, or the commercial object-oriented database management system Gemstone (Graves 1995a, 1995b). A graph database management system that stores data as binary relationships in Unix files has been implemented using Unix C-Shell scripts and the programming language LIFE (Ait-Kaci & Podelski 1991).

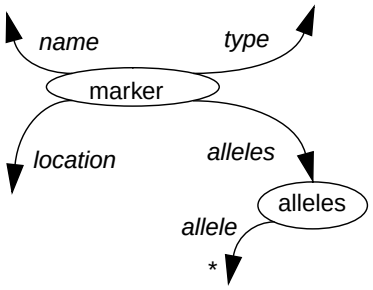
3. Application to Genetics

There are an estimated 50,000 to 100,000 genes in the human genome, but only approximately 2,500 have been located and documented (McKusick 1994). Searching for genes involves creating a map, and placing the genes on the map. A genome map is a description of the order and intervening distance of all the genes in an organism. A genome map has many similarities to a geographic map, such as identifiable locations of genes, distinguishable regions, order of genes on a chromosome, and also many differences, such as genes that jump from one location to another and genes that share a location.

We show that our general process for designing knowledge bases can be used for building a data model including multiple types for genetic markers. We demonstrate the process on a simple representation for marker information in the genome and explain how queries can be asked of the knowledge base. Markers have a physical location on a chromosome which can be identified by some laboratory procedure and whose pattern of inheritance can be followed. Genes are a type of marker which is expressed as a specific functional product, usually a protein. There are other types of markers that are useful in finding the location of genes. For example, one useful type is the *dinucleotide repeat marker*. At many locations in the genome, two nucleotide bases, such as CA, are repeated. The number of repeats can vary between people and the pattern of inheritance of these different *alleles* can be examined within a family to estimate how likely a marker is to be inherited along with a gene of interest. These probabilities can be used to estimate the location of the gene with respect to other markers whose locations are known by converting the probability to a distance using a mapping function (Ott 1991).

There are various factors which make some dinucleotide repeat markers more useful than others, and discovering useful markers is an involved process which can be supported by databases modeling the appropriate laboratory data. However, markers are a foundation for many types of genome maps and must also be represented in knowledge bases which support the automated reasoning and discovery tools which will be more useful in the future.

A marker may be defined as something with a location on a chromosome and a set of alleles. To identify a marker, some information about its type must be known. To define a basic marker for storage in a knowledge base, it can be defined by its relationship to: a name, a location on a chromosome, a marker type, and a set of alleles, as diagrammed below:



Additional definitions are needed for a chromosome location, each allele, and marker types. For many markers, the alleles are differentiated by size (number of nucleotide bases) and the representation of allele can be a set of integers, though a more complex definition is described below. The set of alleles are represented as a separate graph node because of the complexity of information which may be associated with an allele set (especially when multiple populations are being studied).

The representation is roughly the relation:

marker(name, type, location, set of alleles)

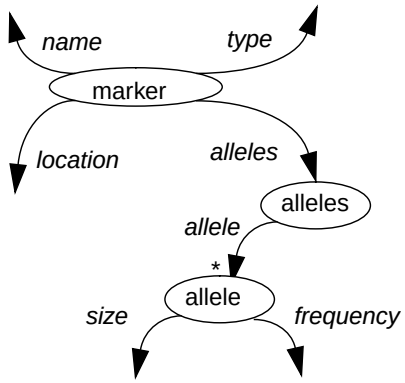
This can be defined using constructive type theory with a data constructor marker which has a type description of:

marker: Symbol x MarkerType x Location x MVA(Allele) --> Marker

where MVA is a type constructor built into SPIDER which treats multi-valued attributes as sets, and Symbol is a built in, unlimited collection of symbols. The other data types are defined in a in a similar manner to Marker.

Sometimes the definition of allele as only a size is not sufficient. Occasionally, we need to know how frequently an allele occurs in the population. Thus, we sometimes -- but not always -- need to store and retrieve frequency information. To avoid redundancy and allow sharing of the information in these two tasks, it is important that the two representations of marker over-

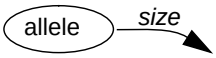
lap as much as possible. Applications where frequency information is important require a graphical representation such as:



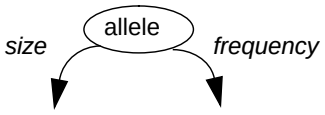
We can use this graphical representation for both relations by having two data constructors for the type Allele, only one of which makes use of the frequency. These definitions are defined by the developer in SPIDER by associating each data constructor with an appropriate WEB graph constructor, here named w<Name>.

defstype Allele ()
    allele1(Size) = wAllele1
    allele2(Size,Frequency) = wAllele2

where wAllele1 is a WEB program which constructs the graph:



and wAllele2 is a WEB program which constructs the graph:



Although it is useful to hide the allele frequencies, the importance of hiding unwanted information in general becomes more clear when considering a more realistic definition of a marker, such as MFD15 (Weber et al. 1990), shown in Figure 2.

Although the representations of marker in Figure 2 may seem clear when examined in isolation, it is important to realize that markers are not particularly useful in isolation but are building blocks for genetic investigations. The use of markers for building blocks require that many markers be viewed in many different ways. One current application is estimating the location of a gene.

Markers are used in locating genes by *typing* the marker against a pedigree (family). Each person in the pedigree is checked to determine which alleles of the marker they have (as shown in Figure 3). Relationships between the typings of several markers and the gene in question can be examined using statistical methods to estimate the location of the gene. Marker typings are also used to estimate the frequency of an allele in a population. A couple of years ago, marker typings might be tested for a few markers over a few hundred people in several pedigrees, e.g. Chamberlain et al. (1993), which leads to a few thousand resulting relations, but one current study is planned to record 800,000

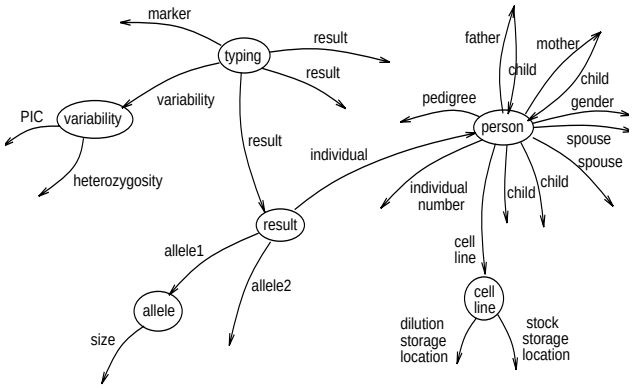


Figure 3: Database view showing one result from a marker typing.

marker typings. It is important to be able to access only the needed information in these studies.

Because we must not only retrieve data using these views, but also define and manipulate it, we cannot just hide data. Graph abstractions must be defined to access each part of the graph directly. These are set up by the knowledge base developer as part of the knowledge base design process.

Abstractions, data constructors, and data types can be generated from a sketch as follows. First, find the sections of the graph which are likely to be reused in a semantically meaningful manner. From Figure 2, the six concepts involved are: marker, locus range, variability, allele set, allele, and reference size. Each of these concepts are associated with a section of the graph. The developer defines a graph constructor to build each section. This leads to the six graph constructors which build the graphs as shown in Figure 4.

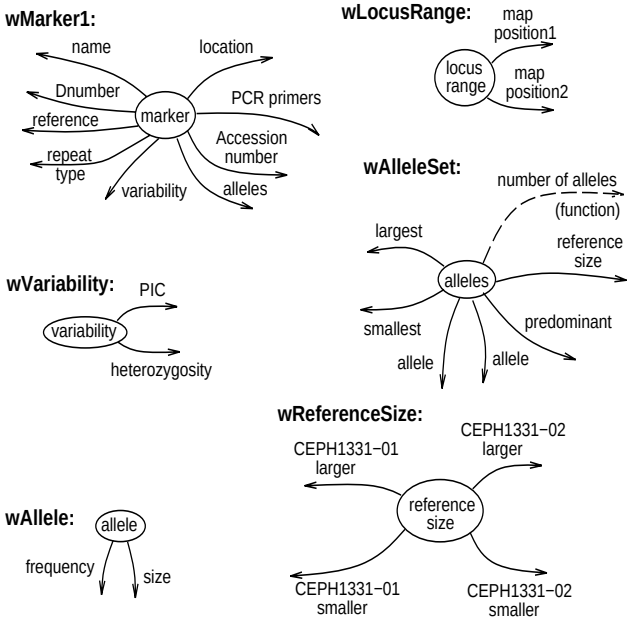


Figure 4: Abstractions for Genetic Markers

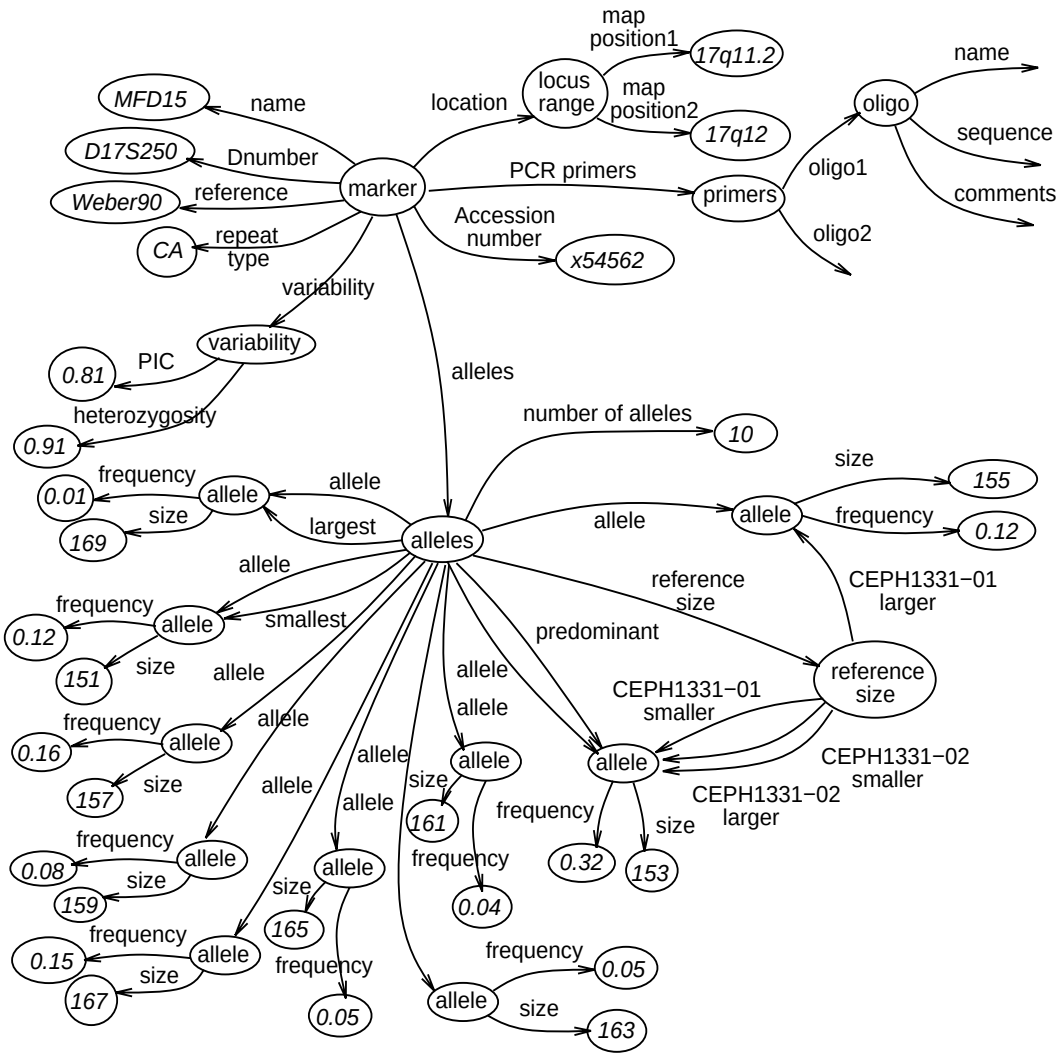


Figure 2: A representation for the CA repeat marker MFD15.

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.

4. Conclusion

We have investigated using a general knowledge base design process to develop application-specific knowledge bases and found that:

1. Knowledge bases may be designed using a process which separates the structure of the data from the behavior necessary for reasoning.
2. Graphs based on attribute value formalisms can be used to represent genetic information.
3. Constructive type theory forms a foundation for data model development which can be incorporated in a knowledge base programming language.
4. A hybrid architecture based on graph logic programming and strongly-typed functional programming can be used to develop prototype knowledge bases for a real-world domain.

The techniques described here have shown themselves to help solve real problems in knowledge base design.

5. Acknowledgments

Thanks to Ellen Bergeman, Karen Mohlke, and Charles Lawrence who commented on earlier drafts of this paper. Thanks to Jeff Chamberlain at the University of Michigan Human Genome Center whose laboratory was used as the basis for the examples in Section 3. The work described here was performed while the author was at the Artificial Intelligence Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan and was supported by NSF grant ISI-9120851. The author is currently supported by an appointment to the Human Genome Distinguished Postdoctoral Fellowships sponsored by the U.S. Department of Energy, Office of Health and Environmental Research, and Administered by the Oak Ridge Institute for Science and Education.

References

1. Ait-kaci H & Podelski A 1991 *Towards a meaning of LIFE*. PRL Research Report 11, DEC Paris Research Lab.
2. Ait-Kaci H 1984. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, Philadelphia, PA.
3. Bic L & Lee C 1987. A data-driven model for a subset of logic programming. *ACM Transactions on Programming Languages and Systems*. 9(4): 618-645, October 1987.
4. Brodie M, Mylopoulos J & Schmidt J, eds. 1984. *On conceptual modelling: Perspectives from artificial intelligence, databases, and programming languages.* Springer-Verlag, New York.
5. Carey M, DeWitt D, Frank D, Graefe G, et al 1986. The architecture of the EXODUS extensible database system. In *Proc of the International Workshop on Object-Oriented Database Systems*. pages 52-65. IEEE, Pacific Grove, CA.
6. Chamberlain J, Boehnke M, Frank TS, Kiousis S, et al 1993. BRCA1 maps proximal to D17S579 on chromosome 17q21 by genetic analysis. *America Journal of Human Genetics*. volume 52, pages 792-798.
7. Cinkosky MJ, Fickett JW, Keen GM 1995. A New Design for the Genome Sequence Data Base. *IEEE Engineering in Medicine and Biology.* Special issue on "Managing Data for the Human Genome Project". (to appear).
8. Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F, and Jeremaes P 1994. *Object-Oriented Development the Fusion Method.* Prentice Hall.
9. Constable RL, Allen SF, Bromley HM, Cleaveland W, et al 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ.
10. Deliyanni A & Kowalski R 1979. Logic and semantic networks. *Communications of the ACM*, volume 22 pages 184-192, March.
11. Graves M 1993. *Theories and tools for designing application-specific knowledge base data models*. Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, University of Michigan. University Microfilms, Inc.
12. Graves M, Bergeman E & Lawrence CB 1995a. A Graph-Theoretic Data Model for Genome Mapping Databases. In the Proceedings of the Hawaii International Conference on System Sciences-28, Biotechnology Computing Track.
13. Graves M, Bergeman E & Lawrence CB 1995b. Graph Database Systems for Genomics. *IEEE Engineering in Medicine and Biology.* Special issue on "Managing Data for the Human Genome Project". (to appear).
14. Kasper RT & Rounds WC 1986. A logical semantics for feature structures. In *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics*, pages 235-242.
15. Lawrence CB 1995. Managing Data for the Human Genome Project. *IEEE Engineering in Medicine and Biology.* Special issue on "Managing Data for the Human Genome Project". (to appear).
16. Martin-Lof P 1982. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy*, pages 153-175, Amsterdam, 1982. North-Holland.
17. McKusick, VA 1994. *Mendelian Inheritance in Man 11th ed.* Johns Hopkins University Press, Baltimore, MD.
18. Nebel B, Smolka G 1990. Representation and reasoning with attributive descriptions. In K.H. Blasius, U. Hedstuck and C.R. Rollinger, editors *Sorts and Types in Artificial Intelligence*, volume 418 of LNAI, pages 112-139. Springer-Verlag.
19. Paulson L 1989. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363-397, September 1989.
20. Prieto-Diaz R & Arango G, eds. 1991. *Domain Analysis and Software Systems Modeling.* IEEE Press.
21. Scriver CR, Beaudet AL, Sly WS, Valle M 1995. *The Metabolic and Molecular Bases of Inherited Disease, 7th ed.* McGraw-Hill, New York.
22. Sowa J 1984. *Conceptual Graphs.* Addison-Wesley.
23. Weber JL, Kwitek AE, May PE, Wallace MR, Collins FS, Ledbetter DH. Dinucleotide repeat polymorphisms at the D17S250 and D17S261 loci. *Nucleic Acids Research*, 18(15):4640, 1990