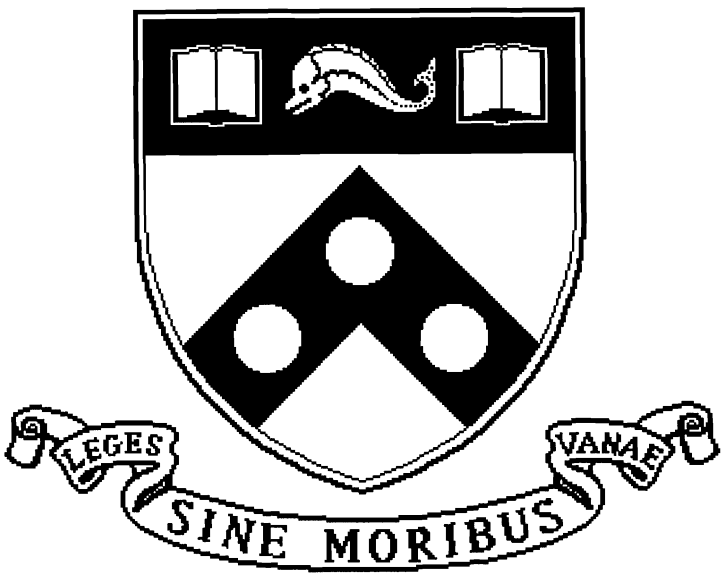
A Data Transformation System for Biological Data Sources

MS-CIS-95-10

P. Buneman
S.B. Davidson
K. Hart
C. Overton



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

February 1995

A Data Transformation System for Biological Data Sources *

P. Buneman, S.B. Davidson, K. Hart, C. Overton
Dept. of Computer and Information Science & Dept. of Genetics
University of Pennsylvania
Philadelphia, PA 19104
Email: {peter,susan,khart,coverton}@cis.upenn.edu

L. Wong
Real-World Computing Partnership
Institute of Systems Science
Novel Function Laboratory
Heng Mui Keng Terrace
Singapore 0511
Email: limsoon@iss.nus.sg

Contact author: Susan B. Davidson, Phone (215) 898-3490, Fax (215) 898-0587

February 24, 1995

Abstract

Scientific data of importance to biologists in the Human Genome Project resides not only in conventional databases, but in structured files maintained in a number of different formats (e.g. ASN.1 and ACE) as well as sequence analysis packages (e.g. BLAST and FASTA). These formats and packages contain a number of data types not found in conventional databases, such as lists and variants, and may be deeply nested. We present in this paper techniques for querying and transforming such data, and illustrate their use in a prototype system developed in conjunction with the Human Genome Center for Chromosome 22. We also describe optimizations performed by the system, a crucial issue for bulk data.

1 Introduction

The goal of the Human Genome Project (HGP) is to sequence the 24 distinct chromosomes comprising the human genome. Much of the information associated with the HGP resides not in conventional databases, but in files that have been formatted according to a variety of conventions. These formats have been adopted in

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, and ARPA N00014-94-1-1086.

preference to database management systems for several reasons. First, the data is complex and not easy to represent in a relational DBMS. Typical structures include sequential data (lists) and deeply nested record structures. This complexity would argue for the use of object-oriented database systems, but these have not met with success because of the constant need for database restructuring [18]. For example, each time a new experimental technique is discovered, new data structures are needed to record details peculiar to that technique. Second, formatted files are easily accessed from languages such as Fortran and C, and a number of useful software programs exist that work with these files. Third, the files and associated retrieval packages are available a variety of platforms.

For example, ACE is an extremely popular data format within the HGP, and has been designed to interact easily with C. Its data model is tree-structure, and allows complex nested types. Many of the schemas that have been developed around it are very intuitive, and easy to understand. A number of sophisticated display and analysis packages have also been developed for ACE, and are available on machines ranging from Sun workstations to Macintosh laptops.

Another popular data format within the HGP is ASN.1 [19]. ASN.1 (Abstract Syntax Notation) which consists of a syntax for types and a prescription of how data conforming to an ASN.1 type is to be physically represented in a sequential file or data stream. It was originally intended as the format for data transport between the top layers of the OSI architecture, but is now being used by the National Center for Biotechnology Information (NCBI) for storing one of the most comprehensive repositories of biological sequence information.

Below we show the specification of an ASN.1 type for the Publication entity in GenBank, one of the databases maintained by NCBI:

```
Publications = {[title: string,
                 authors: {|| [name: string,initial: string] ||},
                 journal: <uncontrolled: string,
                           controlled: <medline-jta: string,        % Medline journal title abbreviation
                                        iso-jta: string,            % ISO journal title abbreviation
                                        journal-title: string,      % Full journal title
                                        issn: string>>              % ISSN number
                 volume: string,
                 issue: string,
                 year: int,
                 pages: string,
                 abstract: string,
                 keywd: {string}]}
```

Rather than use ASN.1 syntax for specifying this type, we have used notation that is close to that of high-level programming languages.

Description	Notation	ASN.1 terminology
list	$\{\|\tau\|\}$	sequence of
set	$\{\tau\}$	set of
record (labeled fields)	$[l_1 : \tau_1, \ldots, l_n : \tau_n]$	sequence
variant (tagged union)	$<l_1 : \tau_1, \ldots, l_n : \tau_n>$	choice

It should be noted that these types can be arbitrarily nested. The variant or "tagged union" is frequently used in this and other formats. Its use can be seen in the example above where journal is either an abbreviated journal name (also a variant type), or the name of the person who performed the data entry (an informal review process).

Query languages associated with data formats are typically very limited. For example, GenBank is accessed through an information retrieval package called Entrez, which simply selects ASN.1 values through pre-computed indexes; no pruning or field selection from values can be performed. Effective query mechanisms for such data, however, must not only be able to extract data, but *transform* data from one format to another. The ability to transform data is not only necessary for manipulating data for storage in archival databases, but for structuring data so that it can be used by other software such as graphical user interfaces and sequence homology packages. Data transformation is also necessary for data integration, in which data from several different sources is integrated into a common format. This is a crucial problem within the HGP since as data sources proliferate, data of interest to scientists is no longer isolated in one or two central data repositories but may be spread across several sources.

We therefore describe in this paper techniques for querying and transforming data that is maintained in these formats as well as data maintained in conventional databases, and illustrate their use on problems arising within the HGP. It should be noted that while our examples deal mainly with ASN.1, the techniques work equally well with a large number of data formats we have studied, including ACE, FASTA, GCG and EMBL as well as object-oriented databases.

The rest of this paper is organized as follows. In Section 2, we describe our model and query language CPL. In Section 3, we give the architecture of a prototype system for querying data formats and databases, and describe how it is currently being used in the Informatics Group of the Center for Chromosome 22 at the University of Pennsylvania. Query optimization is then discussed in Section 4. A brief comparison with other approaches can be found in Section 5.

2 CPL: A Query Language for Collection types

The language CPL (Collection Programming Language) is based on a type system that allows arbitrary nesting of the collection types – set, bag and list – together with record and variant types. The types are given by the syntax

$$\tau := \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \mid \ldots \mid \{\tau\} \mid \{|\tau|\} \mid \langle\!|\tau|\!\rangle \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \mid [l_1 : \tau_1, \ldots, l_n : \tau_n]$$

Here, bool | int | string | ... are the (built-in) base types. The other types are all *constructors* and build new types from existing types. $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ constructs record types from the types $\tau_1, \ldots, \tau_n$. $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ constructs variant types from the types $\tau_1, \ldots, \tau_n$. $\{\tau\}$, $\{|\tau|\}$, and $\langle\!|\tau|\!\rangle$ respectively construct set, bag, and list types from the type τ. An example of this type system, Publication, was given in the introduction.

Data formats also have a syntax for values. Such a syntax is available in CPL as the subset of the language that explicitly constructs values: $[l_1 = e_1, \ldots, l_n = e_n]$ for records; $\langle l = e \rangle$ for variants, $\{e_1 \ldots e_n\}$ for sets; and similarly for multisets and lists. For example, a fragment of data conforming to the Publication type is

```
{[title="Structure of the human perforin gene",
  authors={|[name="Lichtenheld",initial="MG"],
           [name="Podack",initial="ER"]|},
  journal=<controlled=<medline-jta="J Immunol">>,
  volume="143",
  issue="12",
  year=1989,
  pages="4267-4274",
  abstract="We have cloned the human perforin (P1) gene....",
  keywd= {"Amino Acid Sequence", "Base Sequence", "Exons", "Genes, Structural" }]...}
```

This example shows just the first publication record in a set of such records. It is an easy matter to translate from ASN.1 syntax into this format as it is for a variety of other data models. By treating a relation as a set of records, it is also straightforward to represent a relational database in this format. In fact, the type system of CPL (which is slightly larger than the description given here) allows us to express most common data formats including those that contain object identity, which is briefly discussed later. Arrays are also common in data formats, and while they can be expressed as lists, the task of finding the right primitives for array manipulation is an area of current research [16, 24]. We should also remark here that we do not, in general, represent whole databases in this format; it is used for data exchange between the query language of a DBMS or the application programming interface of a data format.

The language CPL. The syntax of CPL resembles, very roughly, that of relational calculus. However there are important differences that make it possible to deal with the richer variety of types we have mentioned and to allow function definition within the language. The important syntactic unit of CPL is the *comprehension*, which can be used with a variety of collection types.

As an example of a comprehension, this is a simple CPL query that extracts the title and authors from a database DB of the type Publication

```
{[title = p.title, authors = p.authors]| \p <- DB}
```

Note the use of \p to introduce the variable p. The effect of \p <- DB is to bind p to each element of the set DB. The use of explicit variable binding is needed if we are to use database queries in conjunction with function definition or *pattern matching* as in this example, which is equivalent to the one above.

```
{[title = t, authors = a]| [title = \t, authors = \a, ...] <- DB}
```

Also, the following queries are equivalent:

```
{[title = t, authors = a]| [title = \t, authors = \a, year = \y...] <- DB, y = 1988}
```

```
{[title = t, authors = a]| [title = \t, authors = \a, year = 1988, ...] <- DB}
```

Apart from the fact that the queries above return a nested structure, they can be readily expressed in relational calculus. The following queries perform simple restructuring:

```
{[title = t, keyword = k]| [title = \t, keywd = \kk, ...] <- DB, \k <- kk}
```

```
{[keyword = k, titles = {x.title| \x <- DB, k <- x.keywd}]| \y <- DB, \k <- y.keyword}
```

The first query "flattens" the nested relation; the second restructures it so that the database becomes a database of keywords with associated titles. Operations such as these can be expressed in nested relational algebra and in certain object-oriented query languages. The strength of CPL is that it has more general collection types, allows function definition and can also exploit variants, which may be used in pattern matching:

```
{[name = n, title = t ]| [title = \t, journal = <uncontrolled = \n> ...] <- DB}
```

This gives us the names of "uncontrolled" journals together with their titles. The pattern <uncontrolled = \n> matches only uncontrolled journals and, when it does, binds the variable n to the name.

The syntax of functions is given by $\backslash x \Rightarrow e$, where e is an expression that may contain the variable x. We can give this function (or any other CPL expression) a name with the syntax define $f == e$ which causes f to act as synonym for the expression e. Thus, the titles of papers of a given author can be expressed as the function

```
define papers_of == \x =>{p| \p <- DB, x <- p.authors}
```

Note that x <- p.authors matches elements of a list rather than elements of a set.

Pattern matching may also be used in function definition, using a vertical bar | to separate patterns:

```
define jname ==     <uncontrolled = \s> =>s
                |   <controlled = <medline-jta = \s>> =>s
                |   <controlled = <iso-jta = \s>> =>s
                |   <controlled = <journal-title = \s>> =>s
                |   <controlled = <issn = \s>> =>s
```

At the risk of some confusion and loss of information, this function finds the identifier or title of a journal. We may use this function in an expression such as

```
{[title=t, name =jname(v)]| [title=\t, journal = \v ...] <- DB}
```

which gives us another example of transforming into a relational database format. A more sophisticated transformation could preserve the tag information from the variant structure in an additional attribute of the relation.

These examples illustrate part of the expressive power of CPL. A more detailed description of the language is given in [8]. An important property of comprehension syntax is that it is derived from a more powerful programming paradigm on collection types, that of *structural recursion* [7, 6]. This more general form of computation on collections allows the expression of aggregate functions such as summation, as well as functions such as transitive closure, that cannot be expressed through comprehensions alone. The advantage of using comprehensions is that they have a well-understood set of transformation rules [51, 45, 44] that generalize many of the known optimizations of relational query languages to work for this richer type system.

Object Identity. While ASN.1 illustrates the complex types typically found in HGP databases, other databases and data formats such as ACE also make explicit use of object identity. For *querying* databases with object identity the type system of CPL is extended with a reference type and the language extended to include a dereferencing operation and a reference pattern. Note that this does not give the language the power to create or update references. For *creating* object-oriented databases, some systems such as ACEDB have a text format for describing a whole database in which the object identifiers are explicit values. We can generate such files with the existing machinery of CPL by applying the appropriate output reformatting routines. For object-oriented

databases that do not have this "bulk load" ability, it is usually an easy matter to make CPL generate the text of a program in native OODB code that calls the appropriate constructors to populate the database.

3 Prototype System and Application in the HGP

Recently, an interesting list of queries thought to be "impossible" in the HGP, primarily due to the lack of tools for querying, integrating and transforming data sources, was published in [13]. An example of one of these queries follows:

Find information on the known DNA sequences on human chromosome 22, as well as information on homologous sequences.

Answering this query requires access to two distinct data sources GDB and GenBank; furthermore, to produce the correct groupings for this query the answer has to be printed as a nested relation. GDB [33, 34]is a Sybase relational database located at The Johns Hopkins University, and is a central repository of map information on physical and genetic maps of all human chromosomes. GenBank [27] is an ASN.1 data source maintained by NCBI, and is accessed through the information retrieval system Entrez. It is located at the National Library of Medicine in Bethesda, MD, and is one the four international repositories for nucleic acid sequence data. To answer this query, GDB is used for obtaining marker information about specified regions (in this case, the whole of chromosome 22), and GenBank is used for accessing precomputed links to retrieve homologous sequences. Other queries in the report also required access to these and other data sources, as well as software systems such as those for sequence analysis (e.g. BLAST and FASTA).

Using CPL, we have developed a prototype system for querying, integrating and transforming data sources within the HGP. Since our intended users are *not* database experts, we have paid careful attention to developing "multidatabase user-views" of the available biological data sources. Multidatabase user-views are not simple integrations of underlying databases (as discussed, for example, in [40, 29, 4, 39]), but represent generalized intended uses of the collection of underlying data sources and frequently involve restructuring data from several sources to some desired format. These user-views are frequently parameterized and programmed with special purpose GUIs such as the one shown in Figure 1, an interface which generalizes the sample DOE query given earlier by allowing users to specify a chromosome and band region of interest.[1] Underlying this simple interface is a CPL function which is executed using the specified parameters.

The overall architecture of the system is shown in Figure 2. CPL is implemented on top of an extensible query system called *Kleisli*[2], which is written entirely in ML [25]. Routines within Kleisli manage optimization, query evaluation, and I/O from remote and local data sources. Once registered in Kleisli, the data drivers perform the task of logging into a specific data source (Open), sending queries in the native form for that source (Query), returning results to Kleisli in internal Kleisli value syntax, and logging out from a specific data source (Close).. For example, a query against the form in Figure 1 would generate a query request to the Sybase driver, which would then access GDB and transform the resulting relation into an internal Kleisli data value; the result of this would then generate input requests to the ASN.1 driver, which would then access GenBank and transform the resulting ASN.1 data value into an internal Kleisli data value. Because communication with the drivers is facilitated through UNIX pipes, drivers can be written in any language; we have used C, perl, and prolog. In addition, a flexible printing routine in CPL allows data to be converted to a variety of formats for use in

[1] This executable screen is available via Mosaic using `http://www.cis.upenn.edu/~khart/form1.html`.

[2] The system is named after the mathematician H. Kleisli, who discovered a natural transformation between monads. This transformation plays a central role in the manipulation of sets, multisets and lists in our system.

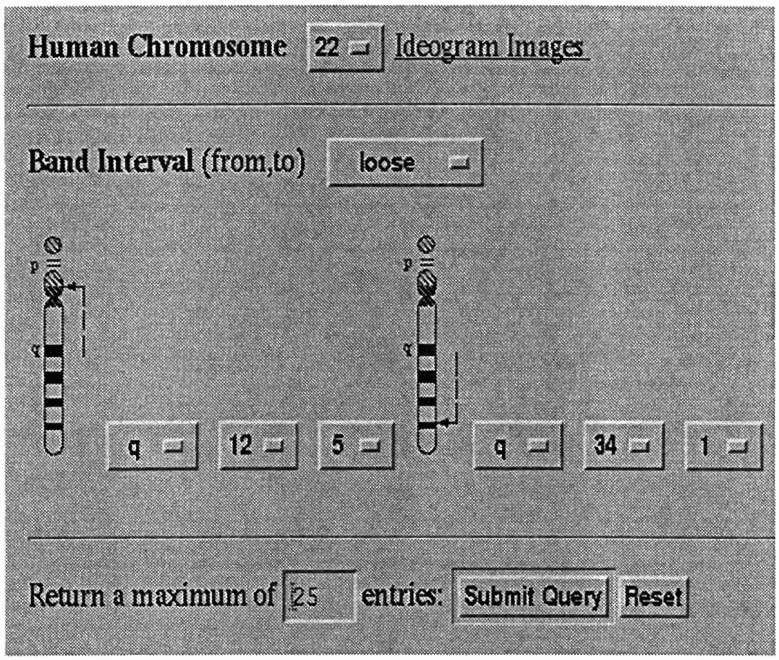


Figure 1: Sample View Interface

displaying (e.g. HTML) or reading into another programming language (e.g. perl).

Kleisli has two interfaces: the application programming interface and the compiler interface. The application programming interface consists of ML modules implementing the data types supported in the model described in the previous section, as well as for token streams and functions. Token streams are important for passing data between CPL and the underlying data sources, and provide Kleisli the mechanisms for laziness, pipelining and fast response. The compiler interface supports the rapid construction of query languages, as we have done for CPL in the present prototype, and contains modules which provide support for compiler/interpreter construction activities. This includes: (1) A general polymorphic type system which supports type unification and type inferencing. (2) An abstract syntax structure for expressing Kleisli programs. (3) A rule-based optimizer and rewrite rule management. (4) A facility for registering external functions. (5) A facility for registering data drivers.

To give an idea of how drivers are used from within CPL, we show two queries accessing GDB (Sybase) and GenBank (ASN.1). These will then be used to implement the sample DOE query.

Querying a Sybase Database. Once a Sybase driver has been registered, driver functions can be used as primitives in CPL to access any relational Sybase database. For example, the following CPL code opens a session with GDB, and defines a function Loci22 which ships an SQL query to GDB. We shall shortly see how the rather complex SQL query is actually generated from CPL by the optimizer in Kleisli.

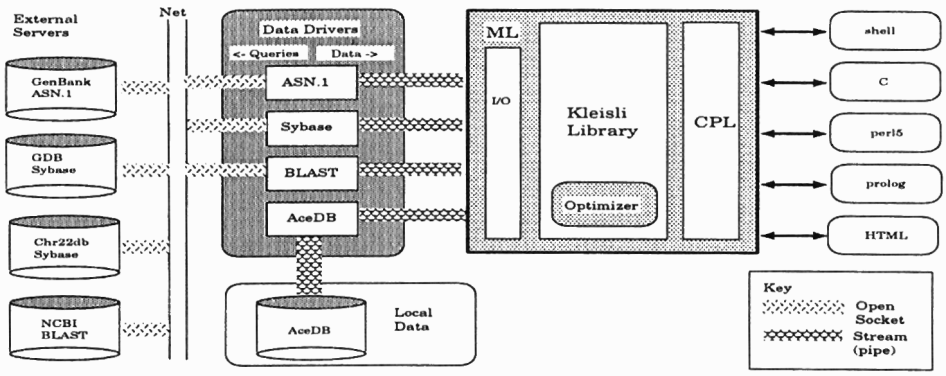


Figure 2: Accessing Biomedical Databases

```
define GDB == Open-Sybase([server="GDB",user="cbil",password="bogus"]);
define Loci22 == Query-Sybase([session=GDB, query="select locus_symbol, genbank_ref
                                                    from locus, object_genbank_ref, locus_cyto_location
                                                    where locus.locus_id = locus_cyto_location_id
                                                    and locus.locus_id = object_genbank_eref.object_id
                                                    and object_class_key = 1 and loc_cyto_chrom_num = '22' "]);
```

In this example, the user has completely specified the query in SQL. However, Kleisli understands how to move selections, projections as well as joins from CPL into Sybase queries. Using an SQL-template function `GDB_Tab` as follows

```
define GDB_Tab == \Table ⇒Query-Sybase([session=GDB, query="select * from " ^ Table]);
```

(^ denotes string concatenation) the previous query could have been written entirely within CPL:

```
define Loci22 == {[locus_symbol= x, genbank_ref= y]|
        [locus_symbol=\x,locus_id=\a, ...] <- GDB_Tab("locus"),
        [genbank_ref=\y,object_id=a,object_class_key=1, ...] <- GDB_Tab("object_genbank_eref"),
        [loc_cyto_chrom_num="22",locus_cyto_location_id=a, ...] <- GDB_Tab("locus_cyto_location")};
```

The optimizer migrates not only all selections and projections to the Sybase server, but also moves the local joins to joins on the server where pre-computed indexes and table statistics may be exploited. Thus, although the second version of Loci22 appears to send three queries to the Sybase server and perform the join within CPL, the optimizer would reconstruct it as in the first version, resulting in a single SQL being shipped.

Querying an ASN.1 Database. The ASN.1 driver for Entrez [27, 28] is significantly more complicated than the Sybase server because there is no real query language interface for ASN.1. While Entrez queries allow the selection of a complex value from an ASN.1 source, they do not allow any pruning or field selection from that value. For example, if the value were set of tuples (a relation), there would be no way to project over certain fields. Although such pruning could be done to an ASN.1 value after it has been retrieved into the CPL environment, we are able to minimize the cost of parsing and copying ASN.1 values by pruning at the level of the ASN.1 driver. For this purpose, we have developed a path extraction syntax that allows for a terse description of successive record projections, variant selections, and extractions of elements from collection.

The selection of ASN.1 values from Entrez is accomplished through pre-computed indexes in the style of information retrieval systems. For the ASN.1 driver, a simple syntax that uses boolean combinations of index-value pairs is used.

To illustrate use of the ASN.1 driver functions, suppose we want the following information:

Retrieve equivalent identifiers corresponding to the accession number `M81409`.

```
define GenBank == Open-ASN([server="NCBI",user="cbil",password="bogus"]);
define ASN-IDs == \accession ⇒
    Query-ASN([session=GenBank, db="na", select= "accession " ^ accession, path="Seq-entry.seq.id..giim", args=[]]);
ASN-IDs("M81409");
```

The driver responds to the ASN-IDs("M81409") query by sending the index lookup select="accession M81409" to the nucleic acid division in Entrez (db="na" – this division contains GenBank), which returns the entries with accession number M81409. The path expression is applied during the parse so that only the ASN.1 sequence ids are returned. In this query, the path expression specifies two projections (.seq.id) on the root type Seq-entry, followed by a variant extraction for each element in the resulting set (..giim). The CPL type specification

```
Seq-entry:[seq:[id:{<giim: int, ...>}, ...], ...]
```

shows the nested types that are encountered by this traversal.

As with the Sybase driver, optimization rules to push projections on ASN.1 data from CPL to Entrez have been written. Although general rewrite rules for the translation of CPL queries to path expressions are not available, we are currently investigating type inferencing for path expressions in order to provide such a translation.

Revisiting the "Impossible" DOE Query. We are now in a position to put the pieces together and answer the DOE query given in the introduction to this section. But first we need to address the issue of homology search.

Loci22 returns information about markers on chromosome 22. To find homologous sequences for these entries we have two choices. We can either extract the nucleic acid sequence from each entry and use it to query a homology search application program (e.g. BLAST or FASTA), or we can use pre-computed similarity links available in Entrez. The homology programs take a query sequence and parameters for the search algorithm, perform the search over a specified database, and return sequence identifiers and match characteristics for each match in that database. The pre-computed links for each entry in Entrez is a list of other entries in the database that were found to have high homology (using BLAST). Since we are only considering nucleic acids in GenBank for this query, use of these links is appropriate and much faster than querying an application program. The

ability to retrieve these links is built into the ASN.1 driver and made available in CPL with the function NA-Links. NA-Links takes a genbank_ref identifier and returns a set of records containing an identifier and a short description of every linked entry.

The final solution to our query can now be expressed using our functions for retrieving markers from GDB (Loci22), translating sequence identifiers (ASN-IDs), and retrieving homologous sequence information (NA-Links). This query is written as

```
{[locus=locus, homologs=NA-Links(uid)]| \locus <- Loci22, \uid <- ASN-IDs(locus.genbank_ref)}
```

Note that the query itself is quite simple, and that most of the effort was spent figuring out where the relevant data was stored.

4 Query Optimization

Optimization of queries is done entirely at compile time using rewrite rules. Rewrite rules are expressed by pattern matching on Kleisli's abstract syntax objects. Thus a rewrite rule R is a function which maps an abstract syntax object to a list of equivalent objects. If R(E) produces $\{E_1, \ldots E_n\}$, then each of $E_1, \ldots, E_n$ is a legal substitute for E. Once such a rule is registered with the rule manager, the optimizer will take it into account when optimizing subsequent queries. In the previous section, we mentioned two families of such optimizations: Pushing projections, selections and joins from the CPL query to the Sybase query, and pushing projections and variant analysis from CPL to the ASN driver. In fact, if any relational subquery in CPL only uses relations from the same database and does not use powerful operators, our optimizer is able to push the entire subquery to the server [49].

Optimizing Joins in CPL. We have also optimized joins performed within CPL by introducing two join operators as additional primitives to the basic Kleisli system. One of them is the blocked nested-loop join [21]. The other is the indexed blocked-nested-loop join where indices are built on-the-fly; this is a variation of the hashed-loop join of [26]. Both operators have a good balance of memory consumption, response time, and total time behaviors. We use the former for general joins and the latter when equality tests in join conditions can be turned into index keys. These two operators are accompanied by 23 optimization rules to help the optimizer decides when to use them. As our system is fully compositional, the inner relations for these joins can sometimes be subqueries. To avoid recomputation, we have also introduced an operator to cache the result of selected subqueries on disk. This operator is accompanied by 3 optimization rules to help the optimizer to decide what to cache.

Lazy Query Evaluation. The evaluation mechanism of Kleisli is basically eager, with rules used to introduce a limited amount of laziness in strategic places to minimize memory consumption and reduce response time. This strategy is the opposite of fully lazy systems which execute lazily by default and rely on sophisticated strictness analysis to bring in eagerness to improve performance [2, 5]. As an example of how lazy evaluation[17, 46] is introduced into our system, consider the nested-loop query

```
{(x, y)| \x <- DB, \y <- S(x)}
```

Note that y is instantiated to members of the set obtained by applying S to x and is thus dependent on x.

Although full evaluation of the query will require instantiating all x and y, each (x, y) pair in the result can be assembled by retrieving a single element x from DB and single element from the set S(x). Where possible, the Kleisli optimizer will lazily retrieve elements from DB and lazily evaluate the function S in order to generate initial output quickly, and minimize storage of intermediate results such as the instantiations of x and y. This mechanism is primarily used when DB and S(x) are derived from external data sources.

Optimizing Projections. We also improve the speed of record projection by exploiting homogeneity. Consider the innocent-looking query below:

```
{[name=n, age=a, sex=s]| [name=\n, age=\a, ...] <- DB1, [name= n, sex=\s,...] <- DB2}
```

This query essentially joins DB1 and DB2. However, we have to compile it with only the knowledge that DB1 has a name field and an age field, and that DB2 has a name field and a sex field. We do not know what are in these fields and we do not know what other fields are present.

Since we cannot compile queries using traditional techniques, which require precise knowledge of types to calculate field offsets at compile time, we have adopted a technique due to Remy[37] (which is related to the extendible technique of Fagin[15]). His technique is to represent a record as a pair consisting of a pointer to a directory and an array. The array keeps the values of the fields of the record. The directory is used to generate the right index into the array given a field name. All records having the same fields share the same directory.

The technique works across systems based on parametric polymorphism [31, 47, 7, 36, 37, etc.] and systems based on subtype polymorphism [9, 10, 11, etc.]. However, not every system needs this kind of generality in record projection. In particular, relational databases have homogeneous sets. In this case, it is possible to take advantage of homogeneity to speed up record projection. To do so, we note that Remy record projection consists of two steps. The first step is the computation of an offset based on field name and the magic number associated with a Remy directory. The second step uses the offset to index into a Remy record to retrieve the value of the required field. If the set we are mapping over is homogeneous, then all its records share the same Remy directory. Therefore, we can apply the idea of code motion [3] and compute the offset only for the first record. This offset can be reused for the remaining records. Our system is able to perform this code motion automatically. A greater than two-fold improvement has been obtained over the plain Remy projection; a full description of the Remy technique and our improvement can be found in [50].

We are also exploiting parallelism both at the data servers and within CPL; for details see [52].

5 Conclusions

In this paper, we have described techniques for querying and transforming complex data types. The language on which it is based, CPL, has been implemented on top of an extensible query system called Kleisli, and is currently being used in the Center for Chromosome 22 at the University of Pennsylvania. Its strengths lie in its ability to represent and manipulate complex data types, and its ability to exploit a "lowest common denominator" of data formats for communication with other data sources. In addition, it is capable of exploiting additional access paths or query languages when these exist, and allows optimizations to "migrate" to these external systems.

The examples we used in this paper showed the system's ability to integrate ASN.1 and relational formats, and

to perform optimizations for these data sources. The techniques work equally well with other data formats, including ACE and a number of interfaces for applications programs. ACE contains certain object-oriented features, specifically classes and object identities. Only minor extensions to the language are needed to query and transform such structures.

Related Work. Issues of integrating databases are not new, and have been dealt with extensively in the computer science literature (see, for example, [43, 42, 23, 30, 48]). The chief distinction between our approach and these is the complexity of data types that we model and query, and the ability to transform between complex types. Although the model in [1] encompasses many of the types we consider (sets, records and variants), the transformations considered are limited and queries are not supported. Our approach also contrasts with that taken by [32] which has a very simple data model and expresses types dynamically. When dealing with biological data sources, static type information is both available and useful in specifying and optimizing transformations.

In the biological domain, the main integration efforts have been either to produce centralized repositories [38], provide indexed or hypertext links between data sources [14, 20], or GUIs to provide fixed integrated access [35, 41]. However, none of these are supported by a query language which allows data to be combined from multiple, heterogeneous sources.

To simplify the specification of complex transformations between databases, we have also developed a declarative language called TSL (Transformation Specification Language), which is based on Horn clause logic (see [12, 22] for details). While it is not as computationally expressive as CPL, TSL naturally captures the structural manipulation of complex data types found in transformations as well as constraints. Having a unified formalism for transformations and constraints is important since there is a significant level of interaction between the two. Once a transformation is specified in TSL, it can be translated into CPL for implementation. However, we have found that using TSL rather than CPL as the specification language considerably simplifies the problem of modifying transformations as schemas evolve. Schema evolution may occur as frequently as every 6 months with biological data sources, and is therefore a significant problem within the HGP. This rapid evolution occurs because as new experimental techniques are discovered, new data structures may be needed to record the details peculiar to that technique.

References

[1] ABITEBOUL, S., AND HULL, R. IFO: A formal semantic database model. *ACM Transactions on Database Systems 12*, 4 (December 1987), 525–565.

[2] ABRAMSKY, S., AND HANKIN, C., Eds. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, Chichester, England, 1987.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] BATINI, C., LENZERINI, M., AND NAVATHE, S. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys 18*, 4 (December 1986), 323–364.

[5] BJORNER, D., ERSHOV, A. P., AND JONES, N. D., Eds. *Partial Evaluation and Mixed Computation.* North-Holland, 1988. Proceedings of IFIP TC2 Workshop, Gammel Avernaes, Denmark, October 1987.

[6] BREAZU-TANNEN, V., BUNEMAN, P., AND NAQVI, S. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece* (August 1991), Morgan Kaufmann, pp. 9–19. Also available as UPenn Technical Report MS-CIS-92-17.

[7] BREAZU-TANNEN, V., BUNEMAN, P., AND WONG, L. Naturally embedded query languages. In *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992* (October 1992), J. Biskup and R. Hull, Eds., Springer-Verlag, pp. 140–154. Available as UPenn Technical Report MS-CIS-92-47.

[8] BUNEMAN, P., LIBKIN, L., SUCIU, D., TANNEN, V., AND WONG, L. Comprehension syntax. *SIGMOD Record 23*, 1 (March 1994), 87–96.

[9] CARDELLI, L. Amber. In *LNCS 242: Combinators and Functional Programming.* Springer-Verlag, 1986, pp. 21–47.

[10] CARDELLI, L. A semantics for multiple inheritance. *Information and Computation 76*, 2 (1988), 138–164.

[11] CARDELLI, L., DONAHUE, J., JORDAN, M., KALSOW, B., AND NELSON, G. The Modula-3 type system. In *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, January 1989), pp. 202–212.

[12] DAVIDSON, S. B., KOSKY, A. S., AND ECKMAN, B. Facilitating transformations in a human genome project database. In *Proc. Third International Conference on Information and Knowledge Management (CIKM)* (December 1993), pp. 423–432.

[13] DEPARTMENT OF ENERGY. *DOE Informatics Summit Meeting Report*, April 1993. Available via gopher at `gopher.gdb.org`.

[14] ETZOLD, T., AND ARGOS, P. Transforming a set of biological flat file libraries to a fast access network. *Computer Applications in the Biosciences 9*, 1 (1993), 59–64.

[15] FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems 4*, 3 (1979), 315–344.

[16] FEGARAS, L. Towards an effective calculus for object query languages, 1995. To appear at the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1995.

[17] FIELD, A. J., AND HARRISON, P. G. *Functional Programming.* Addison-Wesley, Wokingham, England, 1988.

[18] GOODMAN, N., ROZEN, S., AND STEIN, L. Requirements for a deductive query language in the MapBase genome-mapping database. In *Proceedings of Workshop on Programming with Logic Databases, Vancouver, BC* (October 1993).

[19] ISO. *Standard 8824. Information Processing Systems. Open Systems Interconnection. Specification of Abstraction Syntax Notation One (ASN.1)*, 1987.

[20] JACOBSON, D. Prot-web and bioweb—networking for biologists. In *DOE Human Genome Program Contractor-Grantee Workshop IV* (Santa Fe, NM, November 1994), Department of Energy, p. 206.

[21] KIM, W. A new way to compute the product and join of relations. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1980), pp. 179–187.

[22] KOSKY, A. S. A language for database transformations and constraints, 1993. Manuscript available from `kosky@saul.cis.upenn.edu`.

[23] LITWIN, W., AND ABDELLATIF, A. Multidatabase interoperability. *IEEE Computer 19*, 3 (December 1986), 10–18.

[24] MAIER, D., AND VANCE, B. A call to order. In *Proceedings of 12th ACM Symposium on Principles of Database Systems* (Washington, D. C., May 1993), pp. 1–16.

[25] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, 1990.

[26] NAKAYAMA, M., KITSUREGAWA, M., AND TAKAGI, M. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of Conference on Very Large Databases* (1988), pp. 468–478.

[27] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. *ENTREZ: Sequences Users' Guide.* National Library of Medicine, Bethesda, MD, 1992. Release 1.0.

[28] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. *NCBI ASN.1 Spécification.* National Library of Medicine, Bethesda, MD, 1992. Revision 2.0.

[29] NAVATHE, S., ELMASRI, R., AND LARSON, J. Integrating user views in database design. *IEEE Computer 19*, 1 (January 1986), 50–62.

[30] NAVATHE, S., S.GALA, GEUM, S., KAMATH, A., KRISHNASWAMY, A., SAVASERE, A., AND WHANG, W. A Federated Architecture for Heterogeneous Information Systems. December 1989.

[31] OHORI, A., BUNEMAN, P., AND BREAZU-TANNEN, V. Database programming in Machiavelli, a polymorphic language with static type inference. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Portland, Oregon, June 1989), J. Clifford, B. Lindsay, and D. Maier, Eds., pp. 46–57.

[32] PAPAKONSTANTINOU, Y., GARCIA-MOLINA, H., AND WIDOM, J. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering* (March 1995).

[33] PEARSON, P. The genome data base (GDB), a human genome mapping repository. *Nucleic Acids Research 19* (1991), 2237–2239.

[34] PEARSON, P., MATHESON, N., FLESCHER, N., AND ROBBINS, R. J. The GDB human genome data base anno 1992. *Nucleic Acids Research 20* (1992), 2201–2206.

[35] REED, C., AND MARR, T. GDB/Accessor User Guide. Tech. rep., Cold Spring Harbor Laboratory, 1993. URL: **http:/www.cshl.org/gdbacc**.

[36] REMY, D. Typechecking records and variants in a natural extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages* (1989), pp. 77–88.

[37] REMY, D. Efficient representation of extensible records. In *Proceedings of ACM SIGPLAN Workshop on ML and its Applications* (1992), P. Lee, Ed., pp. 12–16.

[38] RITTER, O. The IGD approach to the interconnection of genomic databases. In *Meeting on the Integration of Molecular Biology Databases* (Stanford University, Stanford CA, August 1994).

[39] SHETH, A., AND LARSON, J. Federated database systems for managing distributed heterogeneous and autonomous databases. *ACM Computing Surveys 22*, 3 (September 1990), 183–236.

[40] SHETH, A., LARSON, J., CORNELLIO, J., AND NAVETHE, S. A tool for integrating conceptual schemas and user views. In *Proceedings of 4th International Conference on Data Engineering* (1988), pp. 176–183.

[41] SHIN, D.-G. Developing a graphical sql editor for genomic database federation. In *DOE Human Genome Program Contractor-Grantee Workshop IV* (Santa Fe, NM, November 1994), Department of Energy, p. 90.

[42] SMITH, J., BERNSTEIN, P., DAYAL, U., GOODMAN, N., LANDERS, T., LIN, K., AND WONG, E. Multibase — Integrating heterogeneous distributed database systems. In *Proceedings of AFIPS* (1981), pp. 487–499.

[43] TEMPLETON, M., BRILL, D., DAO, S., LUND, E., WARD, P., CHEN, A., AND MACGREGOR, R. Mermaid — a front-end to distributed heterogeneous databases. *Proceedings of the IEEE 75*, 5 (May 1987), 695–708.

[44] TRINDER, P. W. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece* (August 1991), Morgan Kaufmann, pp. 49–62.

[45] TRINDER, P. W., AND WADLER, P. L. Improving list comprehension database queries. In *Proceedings of TENCON'89, Bombay, India* (November 1989), pp. 186–192.

[46] TURNER, D. Miranda: A non-strict functional language with polymorphic types. In *LNCS 201: Proceedings of Conference on Functional Programming Languages and Computer Architecture, Nancy, 1985* (1985), J. P. Jouannaud, Ed., Springer-Verlag, pp. 1–16.

[47] WAND, M. Complete type inference for simple objects. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science* (Ithaca, New York, June 1987), pp. 37–44.

[48] WIDJOJO, S., WILE, D. S., AND HULL, R. Worldbase: A new approach to sharing distributed information. Tech. rep., USC/Information Sciences Institute, February 1990.

[49] WONG, L. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems* (Washington, D. C., May 1993), pp. 26–36. See also UPenn Technical Report MS-CIS-92-59.

[50] WONG, L. An introduction to Remy's fast polymorphic record projection. Technical Report 94-158-0, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, November 1994.

[51] WONG, L. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

[52] WONG, L. The theory, implementation, and application of a modern query language. In *Progress Report on Flexible Storage and Retrieval of Multimedia Information.* Real-World Computing Partnership Institute of Systems Science Novel Function Laboratory, Heng Mui Keng Terrace, Singapore 0511, December 1994. Available from `limsoon@iss.nus.sg`.