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

L. Wong
Real World Computing Partnership Novel Function Institute of Systems Science Laboratory
Heng Mui Keng Terrace, Singapore 0511
Email: {peter,susan,khart,coverton}@cis.upenn.edu, limsoon@iss.nus.sg

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 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 [?]. 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 for 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 [?]. 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

*Appeared in *Proceedings of 21st International Conference on Very Large Data Bases*, Zurich, September 1995. 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.

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.

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).

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

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 precomputed 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

$$\begin{array}{l}\tau := \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \mid \ldots \mid \{\tau\} \mid \{\!|\tau|\!\} \mid \\ \quad \{\!|\!|\tau|\!|\!\} \mid <l_1 : \tau_1, \ldots, l_n : \tau_n> \mid [l_1 : \tau_1, \ldots, l_n : \tau_n]\end{array}$$

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 $\tau_1, \ldots, \tau_n$. $<l_1 : \tau_1, \ldots, l_n : \tau_n>$ constructs variant types from the types $\tau_1, \ldots, \tau_n$. $\{\tau\}$, $\{\!|\tau|\!\}$, and $\{\!|\!|\tau|\!|\!\}$ 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; $<l = e>$ 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 [?, ?]. 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 the example below, which is equivalent to the one above. Note that the ellipsis "..." matches any remaining fields in the DB record.

```
{[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.keywd }
```

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 [**?**]. An important property of comprehension syntax is that it is derived from a more powerful programming paradigm on collection types, that of *structural recursion* [**?**, **?**]. 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 [**?**, **?**, **?**] 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 [?]. 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 from other organisms.

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 [?] is a Sybase relational database located at Johns Hopkins University, and is a central repository of information on physical and genetic maps of all human chromosomes. GenBank [?] 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 of four international repositories for nucleic acid sequence data. To answer this query, GDB is accessed for information about the GenBank identifiers of DNA sequences known to be within the chromosomal region of interest (in this case, the whole of chromosome 22). GenBank is queried to retrieve the sequence entries along with precomputed links to homologous sequences (i.e., sequences with significant similarity to the original). Only links to non-human organisms are selected. 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 [?, ?, ?, ?]), 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.

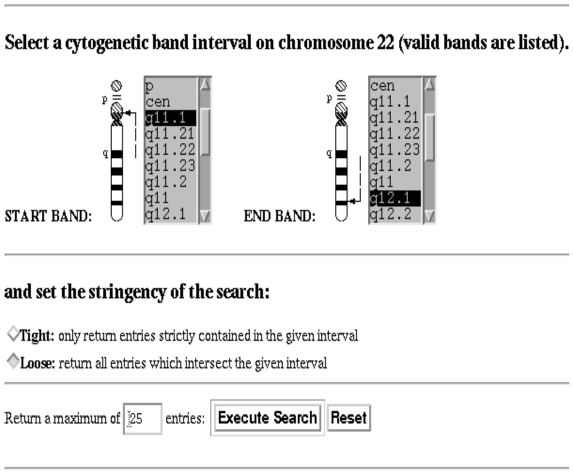

Figure 1: Sample View Interface

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 [?]. 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 in-

[1] This executable screen is available via Mosaic using `http://agave.humgen.upenn.edu/cpl/cplhome.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.

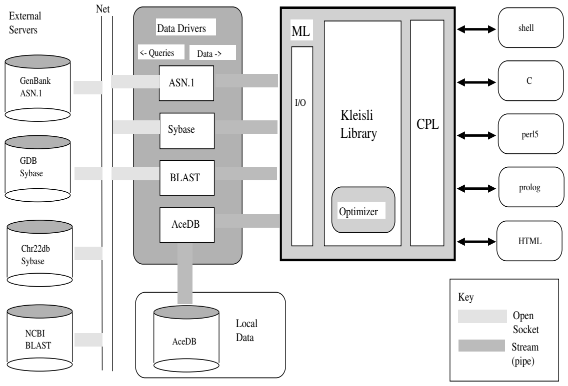

Figure 2: Accessing Biomedical Databases

ternal Kleisli value syntax, and logging out from the data source when the CPL session terminates. For example, a query against the form in Figure 1 would generate a query request to the Sybase driver, which would in turn 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 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.

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.

```
define GDB ==
    Open-Sybase([server="GDB",user="cbil",
                    password="bogus" ]);

define Loci22 ==
    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 =>
    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 query being shipped.

Querying an ASN.1 Database. The ASN.1 driver for Entrez [**?**, **?**] 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 a 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 ⇒
    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, where the ellipsis "..." matches unspecified remaining fields.

As with the Sybase driver, the optimizer migrates projections on ASN.1 data from CPL to Entrez. 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. Loci22 returns accession numbers for known DNA sequences on chromosome 22. ASN-IDs returns ASN.1 sequence ids for given accession numbers. To find homologous sequences for these ASN.1 entries, we use pre-computed similarity links which are available in Entrez via the function NA-Links. NA-Links takes an ASN.1 sequence identifier and returns a set of records describing linked entries. The final solution to our query can then expressed using these functions 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

Crucial to the success of Kleisli for large scale integration of remote data sources is the ability to perform optimizations. Optimization of queries is done entirely at compile time using rewrite rules[3]. The benefit of this approach is that it is easily extensible; new rules can be specified by the designer of the system and grouped into rule sets along with an indication of how they are to be applied, eg. bottom-up or top-down with respect to the tree of subexpressions and how many iterations of a rule set should be applied in what order.

Monadic Optimizations. The core of optimizations in Kleisli is based on rewrite rules derived from the equational theory of monads on which CPL is based. This is similar to relational systems, in which many of the optimizations are based on rewrite rules for the relational algebra, the theoretical basis for relational query languages. Once submitted to Kleisli, a CPL query is translated into an abstract syntax language in the monad algebra NRC to which the rewrite rules can be applied. The monad rewrite rules are initially applied until a normal form is reached; this is guaranteed to terminate in a finite number of steps because the rewrite rules are *strongly normalizing*.

NRC is very similar to CPL, except that it does not use pattern matching and uses the $\bigcup\{e_1 \mid \backslash x <\!\!- e_2\}$ construct instead of the comprehension construct of CPL. The meaning of $\bigcup\{e_1 \mid \backslash x <\!\!- e_2\}$ is the set formed by taking the union of the sets $e_1[o_1/x]$, ..., $e_1[o_n/x]$ where $\{o_1, \ldots, o_n\}$ is the set e_2. Comprehensions in CPL can be translated into this construct of NRC using three simple identities due to Wadler [?] as follows: translate $\{e \mid \}$ to $\{e\}$, $\{e \mid \backslash x <\!\!- e', \Delta\}$ to $\bigcup\{\{e \mid \Delta\} \mid \backslash x <\!\!- e'\}$, and $\{e \mid e', \Delta\}$ to $\mathsf{if}\ e'\ \mathsf{then}\ \{e \mid \Delta\}\ \mathsf{else}\ \{\}$. The last case occurs when e' is not of the form $\backslash x <\!\!- e''$, i.e. it is a boolean expression.

Details of the monad rewrite rules are beyond the scope of this paper (see [?]). We provide only a few of them below. However, the important thing to notice is that they generalize many known optimizations for relational algebra to complex objects.

[3]Run-time optimizations are currently under investigation.

R1 $\bigcup\{e_1 \mid \backslash x <\!\!- \bigcup\{e_2 \mid \backslash y <\!\!- e_3\}\}$
$\leadsto \bigcup\{\bigcup\{e_1 \mid \backslash x <\!\!- e_2\} \mid \backslash y <\!\!- e_3\}$.

Vertical loop fusion is an optimization to combine two loops into one to reduce the amount of intermediate data. It is applicable when the first loop is a producer and the second loop is a consumer. Rule **R1** implements this optimization.

In comprehension syntax this equivalent to:
$\{e \mid \Delta_1, \backslash x <\!\!- \{e' \mid \Delta\}, \Delta_2\}$
$\leadsto \{e[e'/x] \mid \Delta_1, \Delta, \Delta_2[e'/x]\}$
However, Δ_1, Δ_2, and Δ are each irregular sequences of filter and generator expressions, This makes the comprehension form of this rule messy to program. On the other hand, Rule **R1** is trivial to program. This is one of the reasons that the first step in our implementation of CPL is to translate it to NRC.

R2 $\bigcup\{e_1 \mid \backslash x <\!\!- e\} \cup \bigcup\{e_2 \mid \backslash x <\!\!- e\}$
$\leadsto \bigcup\{e_1 \cup e_2 \mid \backslash x <\!\!- e\}$

Horizontal loop fusion is another optimization that combines two loops into one. It is applicable when there are two indepedent loops over the same set, where instead of doing the first loop and then the second loop in a process requiring the set to be traverse twice, both loops are performed simultaneously. Rule **R2** implements this optimization. It applies to sets and multisets, but not to lists.

R3 $\bigcup\{\mathsf{if}\ p(y)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \backslash x <\!\!- e\}$
$\leadsto$
$\mathsf{if}\ p(y)\ \mathsf{then}\ \bigcup\{e_1 \mid \backslash x <\!\!- e\}\ \mathsf{else}\ \bigcup\{e_2 \mid \backslash x <\!\!- e\}$

Filter promotion is an optimization that moves a filter test $p(y)$ closer to the generator $\backslash y <\!\!- e'$ of y. It corresponds to migrating a piece of invariant codes out of a loop. Rule **R3** implements this optimization.

R4 $[l = e, \ldots].l \leadsto e$

This rule corresponds to the traditional database optimization that reduces the number of columns in intermediate data. For example, applying Rule **R1** followed by Rule **R4** to
$\bigcup\{\{x.l_1\} \mid$
$\backslash x <\!\!- \bigcup\{\{[l_1 = f(y), l_2 = g(y)]\} \mid \backslash y <\!\!- R\}\}$
gives us
$\bigcup\{f(y) \mid \backslash y <\!\!- R\}$.

This type of rewriting allowed us to push projections, selections, joins from queries specified entirely in CPL to the Sybase driver in the previous section. In fact, it can be proved [?] that our optimizer is able to push any subquery nor involving nested relations and not using powerful operators to the server.

Optimizing Joins. Other optimizations in Kleisli involve introducing new operators rather than merely rewriting expressions within NRC and are hence called "non-monadic" optimizations. The most important of these are dedicated to improving the performance of joins across data sources, that is, joins that cannot be moved to database servers and must be performed locally. To do this, two join operators have been added as additional primitives to the basic Kleisli system: the blocked nested-loop join [?], and the indexed blocked-nested-loop join where indices are built on-the-fly (this is a variation of the hashed-loop join of [?]). The join rule-set is dedicated to recognizing under what conditions to apply which join operator. For example, the indexed join can be used only if equality tests in the join condition can be turned into index keys. Both operators have a good balance of memory consumption, response time, and total time behaviors.

As the system is fully compositional, the inner relation in a join can sometimes be a subquery. To avoid recomputation, we have therefore introduced an operator to cache the result of a subquery on disk. Rules to recognize when the result of an inner subquery can be cached check that the subquery doesn't depend on the outer relation.

Several of the rules for join optimizations require statistics about the size of files, and can therefore only be used when such statistics are available. We have found it problematic to obtain such statistics on the fly from remote sites, and are currently extending the system to use statically stored statistics from commonly used data sources.

Laziness, Latency, and Concurrency. 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 [?, ?]. As an example of how lazy evaluation 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.

Equally important is the ability to introduce parallelism to improve response time. Consider again the nested-loop query above, and suppose **S** is a function that sends x as a request to some remote database and then returns the reply. Rather than sequentially sending values of x to **S**, we should be able to exploit the fact that many data servers can handle several requests simultanenously. Similarly, while our system is waiting for a response from S, it has enough resources to send a new request to S and process the reply to its previous request simultaneously. We have therefore introduced a primitive that retrieves elements from a collection in parallel and returns the union of the results, and implement it by building pre-emptive thread scheduling into our system using Concurrent ML [?]. Again, rules are introduced to recognize when a function accessing a remote database appears in an inner loop.

In introducing such parallelism, we must be careful of two things: First, the server S may only be able to handle a limited number of requests at a time, say five. In this case, we should send five values of x at a time to avoid overwhelming the server. Secondly, each concurrent thread requires resources (such as memory) to be allocated if their output are not consumed quickly enough. Therefore, techniques to automatically adjust the level of concurrency based on the capability of servers and on resource availability are being developed [?].

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[**?**] (which is related to the extendible technique of Fagin[**?**]). 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 [**?**, etc.] and systems based on subtype polymorphism [**?**, 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 compute the offset only for the first record and this offset can be reused for the remaining records. Our system is able to perform this optimization 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 [**?**].

5 Conclusions

Issues of integrating databases are not new, and have been dealt with extensively in the computer science literature [**?**, **?**, **?**, **?**, **?**, etc.]. 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 [**?**] 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 [**?**] 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 [**?**], provide indexed or hypertext links between data sources [**?**, **?**], or GUIs to provide fixed integrated access [**?**, **?**]. However, none of these are supported by a query language which allows data to be combined from multiple, heterogeneous sources. The idea of using list comprehensions to optimize Daplex queries over protein databases has been studied in [**?**].

The system presented in this paper manipulates complex data types, and is currently being used in the Philadelphia Center for Chromosome 22 for querying and transforming multiple, heterogeneous data sources. Many of these sources are not conventional database systems (such as ASN.1), and we have found CPL useful for extending their information retrieval type languages to a general purpose query language. The strengths of CPL lie in its ability to represent and manipulate complex data types, and in its ability to exploit a "lowest common denominator" of data formats for communication with other data sources. CPL has been implemented on top of an extensible query system called Kleisli, in which optimizations can be expressed. Chief among the optimizations currently being used are the ability to exploit additional access paths or query languages when these exist, and the ability to "migrate" optimizations (such as joins) 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.