

BioKleisli: A Digital Library for Biomedical Researchers ^{*}

S.B. Davidson¹, C. Overton², V. Tannen¹, and L. Wong³

¹ Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104

² Dept. of Genetics, University of Pennsylvania, Philadelphia, PA 19104

³ BioInformatics Center and Institute of Systems Science, Singapore 119597

December 10, 1996

Abstract. Data of interest to biomedical researchers associated with the Human Genome Project (HGP) is stored all over the world in a number of different electronic data formats and accessible through a variety of interfaces and retrieval languages. These data sources include conventional relational databases with SQL interfaces, formatted text files on top of which indexing is provided for efficient retrieval (ASN.1-Entrez), and binary files that can be interpreted textually or graphically via special purpose interfaces (ACeDB). Researchers within the HGP want to combine data from these different data sources, add value through sophisticated data analysis techniques (such as the biosequence comparison software BLAST and FASTA), and view it using special purpose scientific visualization tools. However, currently there are no commercial tools for enabling such an integrated digital library, and a fundamental barrier to developing such tools appears to be one of language design and optimization: The data formats and software packages found throughout the HGP contain a number of data types not available in conventional databases, such as lists, variants and arrays; furthermore, these types may be deeply nested. We present in this paper a framework for providing read access to multiple data sources with complex structured data, and illustrate its use in an application called BioKleisli which accesses data sources critical to the HGP. The three primary components of this framework are: (1) a powerful language for querying and transforming complex structured data; (2) an extensible architecture for implementing the query primitives; and (3) optimization techniques that extend many known techniques to these more complex data types.

1 Introduction

A vast amount of information is currently available in electronic form with internet access. The ability to use

this information involves several distinct problems: first, knowing where the information is that pertains to a particular area of interest; second, accessing the information rapidly; third, efficiently integrating, analyzing and potentially transforming the information into a different form; and fourth, viewing the results in an appropriate manner. Within a particular domain of interest, users typically solve the first problem by formally or informally notifying each other of the existence of various data sources through workshops, conferences, publications, registration on community web pages, etc. That is, there is general knowledge of what the various primary data sources are and some level of documentation available on how to access the data sources and retrieve information. There may also be some general agreement on what the best mechanisms are for visualizing various types of data, contributing to a solution to the fourth problem. For example, postscript files are typically presented using ghostview or some similar displaying tool, and 3-D chemical structures are typically viewed using a variety of sophisticated graphical packages.

However, a fundamental barrier exists to solving the second and third set of problems since the format of the data and functionality of access routines for the data can vary dramatically from source to source. While tools exist for combining data from multiple relational databases, they do not extend beyond the “sets of records” type system of relational databases to more general complex types. It is therefore difficult, if not impossible, to use a single language or access mechanism to obtain, combine and efficiently transform data from multiple non-relational sources. This paper addresses these problems by presenting a language for querying complex value databases [5] and beyond, as well as an extensible architecture for implementing the language and connecting to multiple remote data sources and software packages. Fundamental to the success of the system are powerful optimization techniques that extend to this more complex type system many of the known optimization techniques for relational databases. Thus the approach provides the framework needed for developing a query-server servicing a digital library in a domain where the

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

Correspondence to: susan@central.cis.upenn.edu

location and content of data-sources is known and understood.

A prime example of such an application domain is the Human Genome Project (HGP). The goal of the HGP is to sequence the 24 distinct chromosomes that comprise the human genome, producing physical as well as genetic maps. The benefit of such in depth understanding of human genetic makeup is to enable things like targeted drug development and gene therapy. Several primary sources of sequence data and citations to scientific literature publishing the data have been set up under the support of extensive government funding (such as NIH and DOE); a number of smaller databases also exist at centers charged with sequencing a particular human chromosome; and a number of databases containing ancillary information about other model organisms, metabolic pathway information and related information are also available. These databases are described in various journals, such as the *Journal of Computational Biology*, and discussed widely at the relevant conferences; pointers to the web sites of most of these data sources are also collated on various web pages (a useful example is “Pedro’s BioMolecular Research Tools”, www.public.iastate.edu/~pedro/research_tools.html). Thus, the location of most data that is relevant to biomedical researchers is known, and data is accessible by submitting queries by email or by remote login.

In a 1985 National Academy of Sciences report, “Models for Biomedical Research: A New Perspective,” Morowitz et al [20] argue that biological research has reached a point where “new generalizations and higher order biological laws are being approached but may be obscured by the simple mass of data.” The authors go on to propose the creation of a Biomatrix in which data, information and knowledge are combined to provide an integrated view of biology. However, despite the fact that this data is electronically available, the heterogeneity of format and limited information retrieval facilities pose a fundamental barrier to creating the Biomatrix. Biomedical data sources include conventional relational databases with SQL or OPM [17] interfaces, formatted text files on top of which indexing is provided for efficient retrieval (ASN.1 [38, 40]), binary files that can be interpreted textually or graphically via special purpose interfaces (ACeDB [52]), and image databases of molecular and chemical structures. These formats have been adopted in preference to database management systems for several reasons, chief among which is that the data is complex and not easy or natural to represent in a relational DBMS. Typical data types used in these formats include deeply nested records, sets, lists (sequential data) and variants (union types).

An example of a complex type, `Publication`, is shown in Fig. 1. The type is given in a notation we will use throughout the paper, and the meaning of the notation is explained in the table at the bottom of the figure. Note the nesting of a set of keywords (strings) within the `keyword` record field of the `Publication` type, the nest-

```
Publication = { [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,
  keyword: {string}}
```

Notation	Description
$\{\ \tau\ \}$	list
$\{\tau\}$	set
$[l_1 : \tau_1, \dots, l_n : \tau_n]$	record (labeled fields)
$\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$	variant (tagged union)

Fig. 1. Sample complex type.

ing of a list¹ of author records within the `authors` record field, and the use of a variant or “tagged union” type within the `journal` field representing that publications are either controlled journal entries (also a variant type), or uncontrolled entries containing the name of the person who performed the data entry (an informal review process).

While an object-oriented model [16] would support many of the complex types illustrated in this example, at the time that many of the sequence data sources were established relational systems were widely perceived as the most reliable and stable of database products. Object-oriented database products also did not at that point in time (and arguably not even today) adapt well to schema evolutions [25]. Since schema evolution is frequent within HGP data sources, this was problematic. Schemas evolve rapidly when scientific data is being modeled since the data stored depends heavily on the experimental techniques used, and experimental techniques are constantly being improved. Thus researchers either shoe-horned their data into less expressive but well supported relational database systems, or developed their own query (information-retrieval) systems around expressive formats such as ASN.1 and ACeDB. Other reasons for using formatted file systems include the fact that they are accessible from languages such as Fortran and C, and a number of useful software packages exist that work with these files across a wide variety of platforms. Furthermore, scientists commonly operate with limited budgets and have not seen the need to invest in expensive database management systems – until recently, as the sizes of their formatted-files have bumped

¹ A list type is used rather than a set since the order of authors is very important in scientific publications.

up against the limits of their home-grown tools and the level of funding for the HGP has increased.

The use of relational systems to model such complex data has often led to fragmented, unintuitive designs. Several of the major HGP data sources have therefore recently been using an object front-end called OPM (the Object Protocol Model [17]) for their relational databases to create a more intuitive design for users. OPM is a bare-bones object model with the additional notion of a built-in protocol model to represent laboratory experiments and process flows. The most recent release of the Human Genome Database (GDB 6.0), a major source of HGP information, has been implemented in OPM using the OPM schema design tools, which map an OPM schema into a normalized relational schema with constraints. OPM front-ends can be queried using a restricted object query language; queries are then automatically translated into optimized SQL queries for execution.

To meet the challenge of the Biomatrix, several approaches have been taken by researchers associated with the HGP (see [33] for an overview of these approaches): data warehousing, in which data from various data sources are converted, merged, and stored in a centralized DBMS – examples of this are the (relational) Integrated Genomic Database (IGD) [46] and (Prolog) Genobase [42]; hyperlinking approaches, in which links are set up between related information in different data sources to provide browsing capabilities between databases – examples include SRS [23], WebDBget [6], the ExpASY WWW server by Bairoch, and Genome Net; and standardization of the major data sources in a common format. In fact, standardization in a relational format was as recently as only a few years ago widely perceived within the HGP community to be the only complete solution to the problem of the Biomatrix. In a summit meeting report published by the Department of Energy (DOE) in 1993, several data retrieval tasks that involve combining data from different HGP data sources were claimed to be unanswerable “until a fully relationalized [genomic] sequence database is available” [21].

As with any other collection of legacy systems, it is in fact unlikely that creating the Biomatrix by standardizing on the relational model will work even if such an agreement could be reached. First, many researchers are happy with their existing formats and information retrieval tools and simply will not switch; and second, there still remains the problem of interacting with complex analysis tools to add value to the data. To create the Biomatrix, it is therefore necessary to develop a system which can query and transform existing data from one complex format to another. Such a system will also enable new classes of queries to be posed to existing data sources since the information retrieval languages associated with data formats such as ASN.1 and ACeDB are typically very limited. For example, GenBank (a primary source of HGP genetic sequence data) is accessed through an information retrieval package called Entrez, which simply selects data through pre-computed indexes; no pruning (field selection) from data or joins of data can be performed. The existing information retrieval inter-

faces for such data sources must therefore be augmented by a query language in which such transformations can be expressed.

The digital library application presented in this paper – BioKleisli – therefore represents a major step towards enabling the development of the Biomatrix. The language underlying BioKleisli, the Collection Programming Language (CPL), can query and manipulate data maintained in complex data formats as well as in conventional databases. CPL is described in Sect. 2, and is “complete” for the complex types illustrated in Fig. 1 in a sense that will be discussed in Sect. 2.4.

Equally important to the expressive power of CPL is that it is amenable to optimization. It is an interesting historical note that the relational model was initially believed to be a compellingly simple but impractical idea until optimization techniques were developed, especially for the join operator. Optimization techniques are even more important in a distributed environment where network delays can cause intolerable response times. In such an environment, techniques such as pipelining and parallelism should be exploited to reduce latency. These factors have been taken into consideration in the implementation of CPL in a query system called CPL-Kleisli, which is described in Sect. 3.

In Sect. 4 we give a brief background to the biology and data sources associated with the HGP and describe how an application of CPL-Kleisli, called BioKleisli, is being used for informatics support of the HGP. In particular, we show how BioKleisli has been used to answer many of the “impossible” data retrieval tasks described in the DOE summit meeting report. It should be noted that while our examples deal mainly with relational Sybase and ASN.1 data sources, 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. We also describe how a visualization tool called BioWidget is used for developing front-ends in BioKleisli. A brief discussion of how Kleisli relates to other work can be found in Sect. 5.

2 CPL: a query language for collection types

The principal idea that has guided the design of CPL (the Collection Programming Language) is that of *type orthogonality*. This is an unusual approach in the design of query languages, but it is one of the major advances in recent years for programming languages in general. Since query languages are special-purpose programming languages, once their particularities are understood many ideas from general-purpose languages specialize profitably for query languages. This is true for concepts related to type systems, a point of view strongly advocated by Peter Buneman and his associates over the years [41, 9, 13, 8, 12, 11, 10].

According to the type orthogonality principle, the design of a language is structured around its type system. The primitives of the languages are divided in groups according to the type constructs they support. Adding

new type constructs is as easy as adding a set of primitives. The expressive power is achieved through combinations of the primitives. A well-chosen set of primitives will avoid the “add-on” situation often found in language design: upon realizing that there doesn’t seem to be way of programming some desired construct with what was originally proposed, the construct is added as an ad-hoc primitive.

A less obvious benefit of type orthogonality is that choosing primitives can go hand in hand with understanding the basic laws they obey. This is particularly relevant for query languages, where high-level source-to-source transformations are essential for making many queries actually feasible. The transformation laws that govern the primitives can be applied repeatedly and in combination within optimizers, thus exploiting opportunities that high-level customized transformations may miss. This is the case with the so-called *monad* optimizations performed by the Kleisli system in implementing CPL queries, a new and effective approach to optimizing the collection fragment of object-oriented queries [61].

2.1 The type system of CPL

Following the type orthogonality principle, CPL 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

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

Here, `bool` | `int` | `string` | ... are the (built-in) base types. The other types are all *constructors*, and build new types from existing types. $[l_1 : \tau_1, \dots, l_n : \tau_n]$ constructs record types from the types τ_1, \dots, τ_n . $\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$ constructs variant types from the types τ_1, \dots, τ_n . $\{\tau\}$, $\{\!\{\tau\}\!\}$, and $\{\!\{\tau\}\!\}$ respectively construct set, bag, and list types from the type τ . We have already given in Fig. 1 an example of a CPL type that we call `Publication`.

For each data type, we will have two kinds of primitives: *constructors* and primitives for *decomposition*.² By combining constructors we have a syntax for values of any type. Thus it can be argued that CPL includes its own data format for the data it can represent. The syntax for constructing records is $[l_1 = e_1, \dots, l_n = e_n]$; $\langle l = e \rangle$ is used for variants, $\{e_1 \dots e_n\}$ 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",
```

² Sometimes known as *destructors*, an overly aggressive term. “Deconstructors” would be nicer, but this term has found a loftier calling.

```
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 easy 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 identities and arrays, which are briefly discussed later.

Coming to primitives for decomposition, we begin with records which use simple field selection: `p.authors`. Decomposition of variants is trickier and is best incorporated in a user-friendly syntax by pattern-matching, as shown in the next subsection. For the collection types (sets, bags, lists) we use monad operations, as motivated in [8, 12]. The monad operations form the basis of CPL’s implementation language, NRC (the Nested Relational Calculus). A presentation of NRC is beyond the scope of this paper (see instead [8, 12, 1]), but the basic idea is that all three collection types (sets, bags and lists) use “extension” as a decomposition operation. As examples, extension for sets is $\text{ext}(f)\{a_1, \dots, a_n\} = f(a_1) \cup \dots \cup f(a_n)$ where $f : \tau \rightarrow \{\tau'\}$; extension for bags is analogous with “ \cup ” interpreted as bag union in which duplicates are not eliminated, and $f : \tau \rightarrow \{\!\{\tau'\}\!\}$.

While NRC is perfectly suited for optimizations, it is hard to use directly and we instead turn to a more user-friendly syntax inspired by Wadler’s work [55] called the *comprehension syntax*. (See [11] for details of how the comprehension syntax of CPL can be translated it into NRC for implementation.)

2.2 The comprehension syntax

Using comprehensions as the decomposition operation for collections, the syntax of CPL resembles, very roughly, that of the 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.

As an example of a set comprehension, the following 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`. This set is traversed and the elements in the *head* of the comprehension (the portion to the left of `|`) are collected for the result. In the case of bags, duplicates are kept, and in the case of lists the order of traversal is maintained in the result.

The use of explicit variable binding (`\p` as opposed to plain `p`) 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

previous one. Note that the ellipsis “...” matches any remaining fields in the DB record.

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

As another example, the following two queries are equivalent to each other and extract the title and author of publications appearing in 1988:

```
{ [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, or SQL. Going beyond SQL, the following queries perform restructurings:

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

{ [keyword = k, titles = {x.title} | \x <- DB,
  k <- x.keyword } |
  \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 set of titles. Operations such as these can be expressed in nested relational algebra and in object-oriented query languages as represented by OQL [16]. 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.

Although variants are unusual in traditional data models,³ they are widely found in data formats in the biological domain and are extremely important in CPL. In traditional data models, variants can be circumvented to some extent by splitting sets of records up into multiple sets of different types of records. For example, `Publication` could be replaced by two sets of records, the first representing publications in uncontrolled journals and the second representing controlled journals (which in turn needs to be split in `medline` etc). While such representations are used routinely in relational databases, it is interesting to note that they were rejected by NCBI for GenBank. Using variants makes the `Publication` type significantly more intuitive and natural, and they are easily supported within a query language as CPL demonstrates. The object-oriented data model has other ways

³ Several other less traditional data models such as IFO [4] and IQL [2] also note this lapse and include exceptions.

of circumventing variants, based on inheritance but they seem also quite artificial [19].

CPL supports first-class function types, hence higher-order functions, but in this paper we have chosen to present only the first-order fragment, without explicit function types. For this fragment, functions can be regarded as parameterized macros. Note however that this language is still higher-order in the sense embodied in the comprehension construct. The syntax of functions is given by $\lambda x \Rightarrow e$, 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 relating to a given keyword can be expressed as the function

```
define papers_about ==
  \x => {p | \p <- DB, x <- p.keyword }
```

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

```
define journal ==
  <uncontrolled = \s> => [name=s, type="unctrl"]
  | <controlled = <medline-jta = \s>> =>
    [name=s, type="Medline"]
  | <controlled = <iso-jta = \s>> =>
    [name=s, type="ISO"]
  | <controlled = <journal-title = \s>> =>
    [name=s, type="Full"]
  | <controlled = <issn = \s>> =>
    [name=s, type="ISSN"]
```

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.

These examples illustrate part of the expressive power of CPL. A more detailed description of the language is given in [11] and the theory behind it is presented in [12]. An important property of NRC and hence of comprehension syntax is that it is derived from a more powerful programming paradigm on collection types, that of *structural recursion* [7]. 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 provide a user-friendly syntax but are still equivalent and easily translated to NRC which has a well-understood set of transformation rules [61, 54, 53] that generalize many of the known optimizations of relational query languages to work for this richer type system. It is not clear how such optimizations can be realized with structural recursion. Most of the aggregates however can be formalized as *monad algebras* and the optimization rules for monads generalize for these [29], a promising avenue for the optimization of aggregate queries.

2.3 Other data types: arrays and object identities

Although by far the most common data types we have encountered in HGP data sources are arbitrarily nested records, variants, sets, bags and lists, a number of other scientific data formats include arrays (as in netCDF [45]) and object identity (as in ACeDB). We therefore conclude this section with a brief discussion of how these additional data types can be handled in CPL.

Arrays. While no information is lost when arrays are represented as lists, comprehensions on lists do not conveniently express the variety of operations needed on arrays. The problem of finding the right primitives for array manipulation has therefore recently been addressed by a number of researchers [24, 32, 30]. The approach taken by Libkin, Machlin and Wong [30] is to treat multi-dimensional arrays as partial functions of certain finite domains. The basic primitive is an array “tabulation” construct, $[[e \mid i < n]]$, which is essentially a function definition together with an upper bound for the index (which ranges between 0 and $n - 1$). There is also sub-scripting, $A[i]$, and an operation that gives the size of the array, $len(A)$. (This is syntax for one dimension, but there are corresponding constructs for arbitrary finite dimensions.) As an example, the following expression extracts the even-indexed elements of an array A :

$$[[A[2 * i] \mid i < len(A)/2]].$$

A prototype system that supports an extension of CPL with arrays has been implemented [30]. Array operations are available in the latest CPL-Kleisli version as complex object library calls.

Object Identity. Other data sources commonly found within the HGP also make explicit use of object identity. For querying databases with object identity used merely for link traversal (such as ACeDB) the type system of CPL can be extended with a reference type (`ref of`), and the language extended to include a dereferencing operation (`!`), an identity test, and a reference pattern. This approach can be extended for querying object-oriented databases by associating references with classes (hence adding recursive types) and adding a coerce function to move around in an inheritance hierarchy; methods are modeled as external function calls. Interfaces to ACeDB databases and Shore [15] have been built using these techniques [26].

For example, suppose our database contained an `Author` class with information about `name` (type `string`) and the set of `publications` published by the author (type `ref of Publication`). Assuming that the extent of `Author` is `Authors`, the following expression extracts the titles of papers authored by “Podack”:

```
{ [title =!p.title] |
  ref of Author[name = "Podack", publications = \p, ...]
  <- Authors}
```

2.4 Expressiveness of CPL

An important consideration in any language, whether for a relational database or for an object-oriented database, is how expressive it is. For a relational database language – such as SQL – it is important that it be “complete” in the sense that it can emulate the relational algebra. For nested relational databases, the standard is the nested relational algebra (NRA) [3], which CPL can easily be shown to emulate [61]. However, CPL’s type system also contains lists and bags so it can obviously express more than the NRA. The closest comparison to make is therefore with OQL, the query language part of the ODMG proposal [16] to “standardize” object-oriented databases. We try to review briefly here the similarities and differences between OQL and CPL.

The most striking similarity is that both languages use *comprehension syntax*⁴. In the case of CPL this was a conscious design decision, given the connection between monads and comprehensions [55] and the choice of monad operations for query optimization and execution. The fact that O₂SQL [22] and then OQL use comprehension syntax might have been the result of modifying the SQL `select...from...where` syntax into a compositional and referentially transparent language construct.

There are important differences in the data model. CPL’s basic data model consists of just complex values (no cyclic structures, no oids) although, as mentioned above, a simple extension of CPL has been designed that handles fully object-oriented data [26]. On the other hand, OQL’s data model does not include variants (although a proposal for this appears in [19]). Moreover, OQL is schema dependent in the sense that the system internalizes all the schemas; one cannot compile a query without a schema. CPL is schema independent, since the system does not require the presence of a schema to compile a query; instead, it uses type *inference*, especially record polymorphic type inference. This is very useful in a distributed heterogeneous environment where schemas change often or may be only partially known or understood.

Beyond the data model, the support for inheritance polymorphism is limited in CPL, although CPL has several features in common with function languages that OQL does not: parametric polymorphism and higher-order functions. Moreover, CPL’s array query support is more extensive than that of OQL which relies on programming methods in a language binding such as C++.

3 Kleisli

CPL is implemented on top of an extensible query system called *Kleisli*⁵, which is written entirely in SML [36]. Since the system is being used to query existing data

⁴ Of course, there is a difference in the surface syntax, and OQL’s construct groups all the predicates in a final `where` clause. However, it can be shown that this does not diminish the expressiveness.

⁵ The system is named after the mathematician H. Kleisli, who discovered the characterization of monads that plays a central role in the manipulation of sets, multisets and lists in our system.

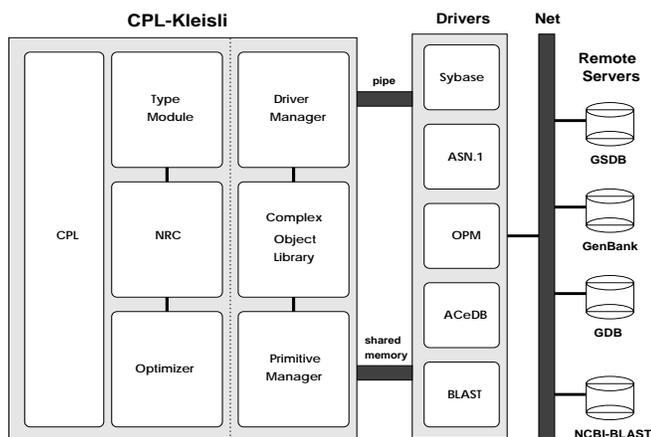


Fig. 2. Architecture of BioKleisli

sources rather than to warehouse data we have not yet found it necessary to consider issues of secondary storage, although this is certainly a consideration for future work. The architecture of an instantiation of Kleisli being used to support the HGP, BioKleisli, is shown in Fig. 2.

Kleisli is extensible in many ways: It could be used to support many other high-level query languages by replacing the CPL module. We call the current prototype *CPL-Kleisli* to distinguish the low-level user interface language being used. CPL-Kleisli could be also used to support many different types of external data sources by modifying the drivers.⁶ The drivers that we have currently implemented (Sybase, ASN.1, ACeDB, BLAST and OPM) are sufficient for connecting to the primary data sources associated with the HGP, and form the basis of BioKleisli. Furthermore, the optimizer of Kleisli can be customized by adding different rules and application strategies.

3.1 The Kleisli modules: high-level language support

The core of the CPL-Kleisli system is divided into two main components, as shown by the dotted line in Fig. 2. The first component provides high-level language support and consists of the CPL module, the type module, the NRC module, and the optimizer. The second component is the query engine and consists of the data driver manager, the primitive manager, and the complex object library.

When a query is submitted to CPL-Kleisli, it is first processed by the CPL module which translates it into an equivalent nested relational calculus (NRC) expression. The abstract calculus NRC is based on that described in [12], and is chosen as the internal query representation because it is easy to manipulate and amenable to machine analysis. It should be noted that the same approach is taken by many relational systems by translation to the relational algebra or to other more convenient

⁶ We use the term “data driver rather than the term “wrappers” [43] since the Kleisli data drivers are relatively simple, merely translating external data types into Kleisli internal format. Any structural transformations of the data are performed within Kleisli.

formalisms. The NRC expression is then analyzed by the type module to infer the most general valid type for the expression, and is passed to the optimizer module. Once optimized, the NRC expression is then compiled by the NRC module into calls to the complex object library. The resulting compiled code is then executed, accessing data drivers and external primitives as needed through pipes or shared memory.

A few comments on the CPL module, the type module, and the NRC module are in order; discussion of the optimizer is deferred to the next subsection.

The CPL module implements CPL by providing a parser for CPL programs and translating them into NRC (see [11] for details). Note that Kleisli can be used to support many different high-level query languages by replacing this module. For example, it would be quite easy to implement SQL using Kleisli by writing a parser for SQL and providing a translation into NRC, although admittedly this would not need the full power of NRC since the type system of SQL (sets of records) and language (relational algebra) is less powerful than that of NRC. It would also be possible to provide an implementation of OQL [16] using Kleisli.

The type module implements a parametric type system that supports record polymorphism as described by [27]. It contains routines for type unification as well as type inference.

To illustrate the type system, let us go back to an example presented in the previous section.

```
define papers_about ==
  \x => {p | \p <- DB, x <- p.keywd}
```

Recall the definition above, where DB is assumed to have type `Publications`. Our system infers that the function `papers_about` has type `string → Publications`, discovering that the variable `x` must have the same type as the element `keywd` field of a publication record and that the output must have the same type as DB.

Now consider a generalization of `papers_about` in which DB is also passed as an input argument:

```
define papers_about2 ==
  [search_key = \x, database = \DB ]
  => {p | \p <- DB, x <- p.keywd}
```

The type module infers that `papers_about2` has type

```
[search_key : 'a, database :
  {[keywd : {'a}; 'b}]] → {[keywd : {'a}; 'b]}
```

The structure of the definition of `papers_about2` imposes some constraints on the structure of the input and these constraints are reflected in the above inferred type. Namely, the `database` field of the input must be a set of homogeneous records having a partially determined type `'b`, which has at least the field `keywd`, which in turn must be a set of objects having an undetermined type `'a`; in addition, the `search_key` field must have the type `'a`;

and lastly, the output must have the same type as the database field of the input.

The 'a above is a *type variable* and it can be instantiated to any type. The 'b above is also a type variable, but it has been constrained so that it can only be instantiated to any record type having the field `keywd`. A type having type variables is called a *polymorphic* type. If a function has polymorphic type, then it can be applied to any input whose type is an instance of the polymorphic type. Our polymorphic type is also *parametric* [35]: All occurrences of the same type variable must be instantiated consistently. That is, if an occurrence of 'a is instantiated to `int`, all other occurrences of 'a must also be instantiated to `int`.

For example, `papers_about2` can be applied to

```
[search_key = "Exons", database = DB]
```

where `DB` conforms to the `Publication` type; in this case the output is also of type `Publication`. The same function could also be applied to an input such as

```
[search_key = 1, database = FOO]
```

where `FOO` is a database of type `{[keywd : {int}]}`. In this case the output also has type `{[keywd : {int}]}`.

There are two points worth noting here. First, `FOO` and `DB` have different types; however, the function `papers_about2` works on both of them, illustrating the power of polymorphism and our ability to write queries that can survive many schema changes of the data sources. It is possible to define in an object-oriented language like C++ a function that acts polymorphically; however, its type must be explicitly declared, as opposed to being inferred. Second, the output type is always the same as the `database` field of the input type, illustrating the accuracy of *parametric* polymorphism. It turns out that it is impossible to achieve this effect in general in an object-oriented language like C++, whose type system suffers from the defect of losing type information [14].

The NRC module provides expression manipulation routines, which are heavily utilized by the optimizer module, as well as routines for compiling NRC expressions into program calls to the Complex Object Library. The expression manipulation routines include useful operations such as testing equivalence of expressions, extraction of free variables, renaming of bound variables, extraction of subexpressions of certain forms, expression substitution, pretty printing, testing for certain properties, etc., which are designed to support the rapid development of new rewrite rules.

3.2 The optimizer

Key to the success of CPL-Kleisli is its ability to optimize. The optimizer consists of an extensible number of phases, each of which consists of a rule base and a rule application strategy. The system currently has six basic rule application strategies such as one that applies

rules in a top-down manner, one that applies rules in a bottom-up manner, and one that applies rules only to maximal redices. It also has several rule bases for optimizations such as filter promotion, pipelining, motion of invariant codes, and migrating joins to external servers. More rule bases and strategies can be inserted to exploit the capabilities of new data drivers and new primitives as they are added to the system.

Rule bases for optimizations can be grouped into two categories: "monadic optimizations", or rewrite rules which fall out from the equational theory of monads on which CPL is based, and "non-monadic optimizations", which introduce new operators rather than merely rewriting expressions within NRC. Some examples of monadic optimizations include:

- Vertical loop fusion, which combines 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.
- Horizontal loop fusion, which combines two loops into one. It is applicable when there are two independent loops over the same set: Instead of first doing one loop and then the other loop in a process requiring the set to be traversed twice, both loops are performed simultaneously.
- Filter promotion, which corresponds to migrating a piece of invariant code out of a loop.
- Reducing the size of intermediate data by reducing the number of fields, which corresponds to moving projections down to intermediate results and base relations in the relational algebra.

Details of these optimizations are beyond the scope of this paper (see [60, 61]), however, the important thing to notice is that the monadic optimizations generalize many known optimizations for the relational algebra to complex objects.

Other non-monadic optimizations include the ability to capture general properties of primitives (such as commutativity) and characteristics of primitives that correspond to relations in external data sources (e.g. the fact that the data source is SQL compliant). These annotations are mostly used in the optimizer in rules having the general form: "If the primitive has property X and if the query expression looks like Y, then optimize the query expression to Z," where "Has property X" is an annotation of the primitive. Such rules that import knowledge of external data sources allows Kleisli to externalize portions of the query processing. For example, the optimizer can be shown to move the largest possible SQL subquery of a CPL expression to an SQL-compliant server, minimizing the amount of data transferred over the net and taking advantage of the powerful commercial optimizers of those systems.

The most important of the non-monadic optimizations 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 [28], and the indexed blocked-nested-loop join where indices are built on-the-fly (this is a variation of the hashed-loop join of [37]). 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. In addition, the optimizer also parallelizes joins involving remote sources to reduce latency.

As the system is fully compositional, the inner relation in a join can sometimes be a subquery. To avoid recomputation, we have therefore also 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 must then check that the subquery doesn't depend on the outer relation.

More complete examples of non-monadic optimizations can be found in [62, 61].

3.3 The query engine

Once optimized, the NRC expression is compiled into calls to the complex object library and executed, accessing data drivers and external primitives as needed.

The Complex Object Library provides routines for the manipulation of sets, bags, lists, records and variants, and forms the core of the query execution engine. The manipulation routines are designed according to the principles of structural recursion [12]. Support is also provided for data parallelism and lazy evaluation.

The Primitive Manager Module This module implements an environment for managing primitives defined within CPL (macros) as well as primitives imported from external systems. For example, external software such as sequence analysis routines and image analysis routines can be used as first-class citizens in CPL once they have been registered with the primitive manager.

The primitive manager maintains the type and CPL definition of all macros, the CPL definition being used for optimization as macros are unfolded within queries. CPL-Kleisli also provides a "materialize" command. When a user materializes a macro, compiled code is produced and recorded by the primitive manager. From then on, the materialized macro behaves like an external black-box primitive.

Since an external primitive is implemented externally and imported into CPL, its type and compiled code must be maintained by the primitive manager. For primitives written in SML, the compiled code can consist of the entire external function definition. Primitives written in other languages, such as C, must be wrapped in an SML wrapper which calls the external software using standard things like the C interface to SML and system calls. The compiled code of this wrapper is then maintained by the primitive manager. Information about where the software is and how it can be run must be available to the

SML wrapper either by hardwiring it into the wrapper code, or obtaining it indirectly from a Unix environment, an external file or as input arguments when the primitive is invoked.

Although no optimizations that require access to the code of external functions can be performed, the primitive manager allows primitives to be annotated with various properties such as whether it is commutative, associative, idempotent, expensive, etc. These annotations can be used by optimization rules to optimize subexpressions involving these primitives. For example, a rule for constant-folding will optimize $((5 + x) + 6)$ to $(11 + x)$, by checking that the imported primitive "+" is annotated to be commutative and associative. Another example of how annotations can be used in optimization was given earlier, one which allowed the largest possible SQL subquery of a CPL expression to be moved to an SQL compliant server.

The Driver Manager Module provides an environment for managing data drivers for external data sources. In the BioKleisli implementation illustrated in Fig. 2, drivers are provided to Sybase, ASN.1-Entrez, OPM and ACeDB databases as well as the BLAST sequence analysis package. To access an external data source, Kleisli submits a request to the corresponding data driver. The driver forwards the request, possibly after some translation, to the data source. When the data source responds, the data driver passes the response back to Kleisli in an agreed upon exchange format. Currently, all our drivers uses the same self-describing data exchange format. New drivers can be added to the system by registering them with the Data Driver Manager.

In contrast to primitives, data drivers have state, e.g. in the form of socket connections, which is also managed by the data manager module. For each data driver, the manager maintains:

1. Its input type.
2. A function to produce the output type of a driver call. The function is provided with the NRC-representation of the input argument to the driver call during type inference (i.e. at compile time). Note that the output type of calls to a driver may vary. For example, in one Sybase call, the query may produce a relation with two columns and in another call the query may produce a relation with three columns.
3. A suggested concurrency level. Since many DBMS are multi-user systems, it is possible to maintain multiple connections against a single server. These connections can be used to introduce parallelism to the execution environment. The driver manager will try to keep as many connections to the drivers active as suggested by this level.

Further details on Kleisli can be found in the reference manuals [59, 58].

The ease with which Kleisli can be modified has proved extremely useful as various extensions to BioKleisli and CPL have been made. Adding new drivers (such as generic Oracle and ACeDB drivers) and adding new

non-monadic optimization techniques (such as a semi-join operator) has been fairly routine. We also intend to introduce statistical optimization techniques to allow re-ordering of joins across multiple databases and to determine which data source to use when data is replicated across several external sources. For example, within BioKleisli the same data is replicated at different data sources. However, the access time for obtaining data and the query capability varies dramatically from source to source: a relational server can perform field selection whereas an ASN.1 server cannot. By experimenting with several stereotypic queries in BioKleisli (see Sect. 4), we have realized substantial performance gains by writing the queries to use the “best” server. We are currently exploring ways of using the results of these experiments by capturing statistics about the underlying data sources (such as the size of complex relations, the selectivity of field selection, and the speed of the servers), and encoding their use as rules during the optimization phase.

We have also found it relatively easy to extend CPL in various ways. One extension is the addition of arrays [30], which entailed defining the primitive operations on arrays (see Section 2) and their associated optimization ruleset. We are now extending CPL so that we can connect to object-oriented databases by defining a new abstract data type – reference – together with a dereference operation and equality.

4 BioKleisli: towards the Biomatrix

BioKleisli is an application of CPL-Kleisli which was developed in 1994 and is currently being used both for informatics support of the Philadelphia Center for Chromosome 22 as well as for the wider HGP community. There are two primary modes of use of the system: through the CPL interface, and through a number of user views.

CPL is a compact, complete language for working with deeply nested complex data types. Due to its close syntactic ties with the relational calculus, most database programmers find it easy to program in. However, most users of BioKleisli are not database programmers and prefer a more intuitive (if limited) point and click type of interface. We have therefore put together a number of stereotypic parameterized queries (or user views) and made them available over the Web. The queries span multiple databases and appear as simple GUIs such as the one shown in Fig. 3.⁷ Implementing these screens involves translation to CPL queries using the input parameters specified by the user.

The stereotypic queries were originally drawn from the list of impossible queries published by DOE, and have been augmented on-demand from the HGP user community. The significant interest in this system is testified to by the tremendous increase in Web hits to our site, which has gone from 19 hits in July 1995 to 683 hits in July 1996.

⁷ This executable form along with those associated with the queries below can be found at <http://agave.hungen.upenn.edu/cpl/cplhome.html>.

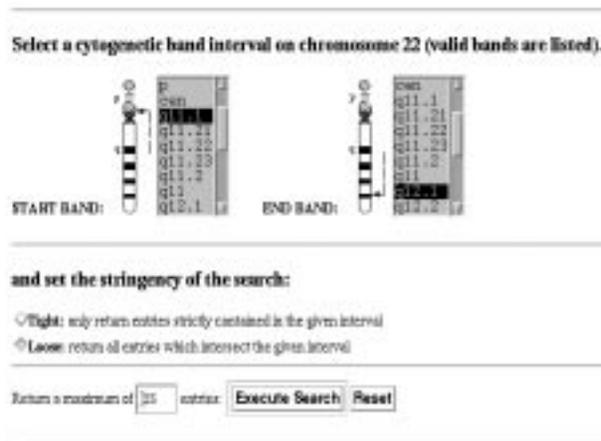


Fig. 3. Sample view interface

To give the flavor of how BioKleisli is being used and why these stereotypic queries are interesting, we start with a brief introduction to the biology behind the queries and the data sources involved. We then outline the stereotypic queries, and describe in detail how one of them is implemented.

4.1 A brief biology lesson

In bacteria and higher organisms, genetic information is recorded in DNA (deoxyribonucleic acid), a long chain polymer composed of units of deoxyribonucleotides having one of the four bases adenine (A), guanine (G), cytosine (C), and thymine (T). DNA is normally found as a double-stranded helix with the two strands running in opposite directions and held together by bonds between complementary A and T, or G and C pairs, where one member of the pair is on each strand. Exactly the same information is contained in each DNA strand, an important property during cell division when the complement of each strand is synthesized to form two copies of the original DNA which are then partitioned into the two new daughter cells. For the purposes of this paper, we can ignore the chemical properties of DNA and its constituent subunits and simply view it as a string of characters over the alphabet A,G,C, and T.

DNA of up to several hundred million characters in length is packaged by structural proteins to form a chromosome. Fig. 4A shows the 24 distinct chromosomes (two sex chromosomes, X and Y, plus twenty-two others) found in humans. Each chromosome has been stained, generating the alternating dark and light bands which are visible under a microscope. The banding pattern for each chromosome is unique, and can be described by a (more-or-less) standard nomenclature adequate for both normal and abnormal chromosomes as shown in the ideogram of chromosome 15 (Fig. 4B).

The most important information recorded in DNA is that for genes, which encode the instructions for building proteins. Proteins are the workhorses of the cell, performing the primary enzymatic and structural functions required for life. Like DNA, they are long-chain

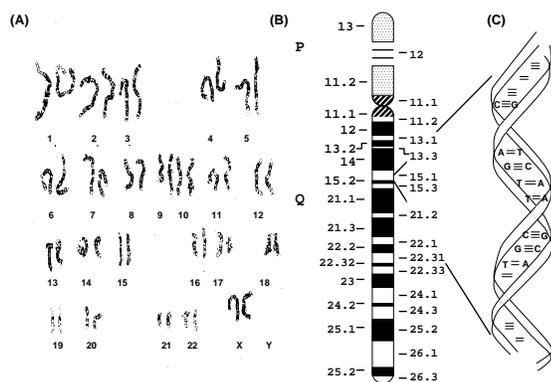


Fig. 4. (A) 23 pairs of normal human female chromosomes labeled 1-22 and X (Y chromosome not shown). Courtesy of Nancy Spinner, Children's Hospital of Philadelphia. (B) Ideogram of the high-resolution banding pattern of chemically stained chromosome 15. Labels indicate the standard nomenclature for stained bands along the short (P) and long (Q) arms of the chromosome. (C) Sketch of a sequence of DNA from chromosome 15 represented as a double helix. A:T pairs are held together by double hydrogen bonds while C:G pairs have triple bonds.

polymers, in this case formed from a set of 20 distinct amino acids. Proteins are synthesized from the information in genes in a two step amplification process depicted in Fig. 5. The first step, *transcription*, copies the information in a gene to an initial RNA molecule (RNA or ribonucleic acid is very similar to single-stranded DNA) which is then edited to form messenger RNA (mRNA) by removal of introns and the addition of a tail of A subunits. The average size of mRNAs in higher organisms is 1200-1500 nucleotides. The mRNA is then *translated* into protein by cellular machinery which reads the sequence in blocks of three nucleotides. The mapping from nucleotide triplets to amino acids is highly conserved throughout all living organisms. Note that in higher organisms the coding regions make up around 5% of the genome, the remaining 95% of the genome is in introns and intergenic DNA.

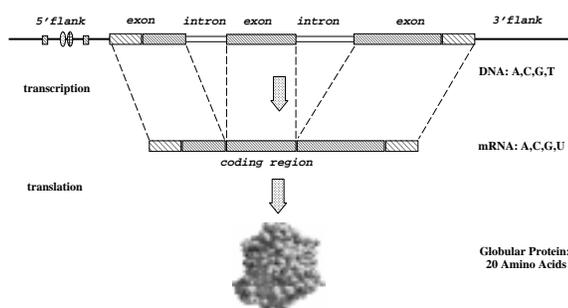


Fig. 5. From DNA to protein: schematic of gene, mRNA and protein structures and transcription and translation processes. The space filling model of β -hemoglobin was rendered using RasMol.

It is the ultimate goal of the HGP to determine the complete three billion long sequence of the DNA contained in the human genome and to sequence the genomes of a number of important model organisms including yeast, the fruit fly *D.melanogaster*, the nematode *C.elegans*, and mouse. *Sequencing* DNA means determin-

ing the exact order of A's, C's, T's, and G's on the string. Although there are techniques for directly sequencing short lengths of DNA (approximately 400 to 500 bases), current methods are not practical for sequencing an entire chromosome at one time. Consequently the HGP set mapping the chromosomes as its initial goal to be followed by large-scale, high-throughput sequencing. *Mapping* involves ordering unique markers along the chromosome to use as landmarks to orient sequencing projects and more immediately to aid in localizing genes. Large-scale mapping projects have already proven extraordinarily successful, leading to the discovery of numerous genes involved in human disease, and preparing the stage for the high-throughput sequencing phase of the HGP which is now underway.

Mapping. A variety of different techniques are used to order markers along the chromosome. These can be broken down into two main categories: genetic and physical mapping. Genetic mapping determines the order and relative distance between markers for observable traits such as those for genetic diseases or, more recently, for variations in DNA sequence among individuals. Physical mapping orders markers based on the characteristics of chromosomes or the underlying DNA without reference to genetic variations among individuals. For example, the banding pattern of chromosomes is a low resolution form of physical mapping. Far higher resolution mapping can be had through techniques using cloned probes and Sequence Tag Sites (STS's). In these approaches, the chromosome of interest is randomly fragmented into overlapping pieces of experimentally manipulable size (50,000-1 million bases). These pieces are then reassembled into a linear ordering representing their order in the original DNA string. To discover the relative ordering of fragments, it is crucial to be able to ascertain when the sequence of two pieces of DNA overlaps, that is, when the pieces come from neighboring sites in the original string. Sequence overlap between two pieces of DNA can be detected by showing that their sequences contain the sequence of a third, much shorter fragment, called a *probe*. The linear ordering on the pieces yields a linear ordering on the probes whose sequence is contained in them, and vice versa. The probes then become the desired map landmarks and may be used to sequence areas of special interest, such as regions thought to be related to inheritable disease.

4.2 BioKleisli data sources

As illustrated in Fig. 2, BioKleisli currently connects to Sybase, ASN.1-Entrez, ACeDB and OPM database systems as well as the BLAST sequence analysis package. This capability, which is still only in its initial stages, provides access to many of the major databases and resources commonly requested in the HGP. For the purposes of this paper, we will focus on the following data sources:

- The community archival nucleic acid sequence data banks, **GenBank**, **EMBL**, **DDBJ**, and **GSDB**. GenBank (National Center for Biotechnology Information (NCBI), National Library of Medicine, USA), EMBL (European Molecular Biology Laboratories, EBI, UK) and DDBJ (DNA Data Bank of Japan, Japan) are part of a worldwide consortium for archiving and curating the DNA and RNA sequence data for all organisms and selected ancillary information. The common exchange and distribution format for sequence data is as a flat file, but each resource maintains its own internal format and access capabilities. GenBank for example uses the data exchange format ASN.1 as their internal data model and has developed a custom information retrieval system, ASN.1-Entrez, for WWW-based access data access. The National Center for Genome Resources (USA) maintains a mirror of GenBank, GSDB, as a Sybase database. GSDB is accessible via the WWW through an SQL interface. While the size of the sequence databases are relatively modest at present, less than 5 gigabytes in flat file format, the rate of growth has been exponentially for over a decade with a doubling time of about 18 months.
- The Genome Database (**GDB** [44]), located at The Johns Hopkins University, is a central repository of information on the physical and genetic map of the human genome. It is the sole integrated source of mapping information but it draws from a number of independent databases on chromosome specific or technique oriented information on mapping. Unlike sequencing data, mapping information is intertwined with specific experimental protocols which has led to considerably more complex data representation requirements. The latest release of GDB (ver 6.0) has been completely reimplemented from a Sybase version to an OPM version. Although it is still possible to query the underlying relational form of GDB 6.0 by SQL, the primary form of access to GDB 6.0 is through an object broker which traffics in the OPM query language.
- **BLAST** is a software package for performing comparisons of a query nucleic acid or protein sequence against a database of sequences (GenBank for instance). It employs a string comparison algorithm tuned for biological sequences. Conceptually, BLAST searches resemble text retrieval system. The output from BLAST is formatted in either a verbose human readable form or as ASN.1 data structures.

The Sybase driver can be used without modification to connect to any Sybase relational database. The driver takes as input any TransactSQL statement (e.g., an SQL query), and returns a CPL set, bag, or list equivalent to the resulting relational table. Generally the SQL commands received by the driver are generated by CPL's optimizer in the course of pushing joins, selections, etc. to the Sybase server. Thus in most cases the SQL statements are guaranteed to be correct. In the remaining cases, the driver supports a protocol by which it can communicate errors back to CPL. Furthermore, it supports

a number of high-level (i.e. not expressed in SQL) metadata queries which allow CPL to retrieve schema information in a manner specific to relational databases, but not specific to any particular relational database. This is because the driver presents an abstract relational view of the metadata. To take full advantage of this feature we have implemented a CPL module which caches relational metadata locally, allowing the system to typecheck user-specified SQL statements prior to passing them to the drivers.

The ASN.1-Entrez driver is used to connect to ASN.1-GenBank [40]. We have developed a similar driver to access BLAST servers. These drivers [40, 38] are significantly more complicated than the Sybase one because there is no real query language interface for ASN.1. The selection of ASN.1 values from Entrez is accomplished through pre-computed indexes in the style of information retrieval systems, i.e. one whose syntax simply uses boolean combinations of index-value pairs. No pruning or field selection from values can be specified; e.g. if the value were a set of records there would be no way to specify a projection over certain fields of the records in that set. Although such pruning could be done to an ASN.1 value after it has been retrieved into the CPL environment, we 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 collections.

The OPM/Object Broker driver is used to connect to any database operating under GDB's Object Broker architecture (i.e. GDB 6.0). The object broker is based on OPM and hence the driver accepts queries in the OPM query language. In a way similar to the Sybase driver, the OPM driver can also perform metadata queries, something which the OPM query language does not support. The driver uses GDB's "OBMETA" library for this purpose, which unfortunately requires local copies of all the schema files, making the consistency issue a problem. However, the Object Broker client libraries do support a mode in which the individual components (OPM objects in this case) of a query result can be processed as soon as they are available. This is ideal for CPL's pipeline-oriented optimization rules, which strive to minimize query response time in addition to minimizing overall query time.

In contrast to the generic Sybase data driver, the ASN.1 and OPM data drivers are much more database specific. The nature of NCBI's ASN.1 libraries makes it very difficult to write source code that is divorced from the schema of the data being accessed. The situation is similar to that for OPM, in which a schema file is read, processed, and stored in some compiled format that must subsequently be linked to by applications querying the database (i.e. OPM's "query translator" module).

Numerous other drivers also exist, including drivers for ACeDB, ObjectStore, and Oracle. A WWW driver is also capable of retrieving documents through the http protocol.

4.3 Answering the “unanswerable” queries

Using BioKleisli, we have put together several stereotypical biomedical database queries and made them available over the Web. Since users of this HGP digital library system are not database experts, we have paid careful attention to developing “multidatabase user-views” of the available biological data sources. Seven major classes of such parameterized queries are currently available, with new ones being added by request from the user community. They were chosen to test system performance in formulating and executing distributed queries while providing functionality to the growing HGP user community. These queries are shown below with the databases accessed indicated in parenthesis.

- Gene/Location query: “Find protein kinase genes on human chromosome 4.” (GDB.)
- Sequence/Size query: “Find mapped sequences longer than 100,000 base pairs on human chromosome 17.” (GDB and GSDB.)
- Mapped EST query: “Find ESTs mapped to chromosome 4 between q21.1 and q21.2.” (GDB and GSDB.)
- Primate alu query: “Find primate genomic sequences with alu elements located inside a gene domain.” (BLAST and GSDB.)
- BLAST Sequence/Feature query: “Find sequence entries with homologs of my sequence inside an mRNA region.” (BLAST and GSDB.)
- Human genome map search: “Find human sequence entries on human chromosome 22 overlapping q12.” (GDB, GSDB and ASN.1 GenBank.)
- Non-human homolog search: “Find information on the known DNA sequences on human chromosome 22, as well as information on homologous sequences from other organisms.” (GDB, ASN.1 GenBank and BLAST.)

To illustrate how BioKleisli is used to answer these queries, we will go through the last query (non-human homolog search) in some detail. The strategy taken is to access GDB 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 then 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. To produce the correct groupings for this query, the answer is printed as a nested relation.

The GDB Query. Since a Sybase driver is registered within BioKleisli, driver functions can be used as primitives in CPL to access GDB. For example, the following CPL code opens a session with GDB, and defines a function Loci22 which ships the given SQL query to GDB. The SQL query joins three tables in GDB (`locus`, `object_genbank_ref` and `locus_cyto_location`) together on common identifier fields, and extracts the `locus_symbol` field from the `locus` table and `genbank_ref` field from the `object_genbank_eref` table. The `locus_cyto_location` table is

used to restrict entries to lie on chromosome 22. We shall shortly see how that this rather complex SQL query is actually generated from CPL by the optimizer in BioKleisli.

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

define Loci22 == GDB([query=
  "select locus_symbol, genbank_ref
  from locus, object_genbank_eref, 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, the monadic optimizations of Kleisli together with some additional rewriting rules can move selections, projections as well as joins from CPL into Sybase queries. In fact, it has been proved [61] that the optimizer is able to push any subquery not involving nested relations and not using powerful operators (such as group-by) that involves a single relational server down to that server.

To write the query entirely in CPL, we first define an SQL-template function `GDB_Tab` for queries against GDB (`^` denotes string concatenation):

```
define GDB_Tab == \Table =>
  GDB([query="select * from " ^ Table ]);
```

The previous query, `Loci22'`, can now be written entirely within CPL as:

```
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 GDB and perform the join within CPL, the optimizer would reconstruct it as in the first version, resulting in a single SQL query being shipped.

The ASN.1 GenBank Query. 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 `"-D"` flag tells the Entrez driver to extract only summary information for records having this accession number. The summary information is a record, one of whose fields (`uid`) is the ASN.1 sequence identifier requested.

Although not shown in this example, the optimizer migrates projections on ASN.1 data from CPL to Entrez as was done with the Sybase driver. 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 non-human homolog search query. We are now in a position to put these two pieces together and answer the non-human homolog search query. `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 be 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.

Performance results. To give an idea of the size of the result of the non-human homolog search query, there are approximately 2,000 locus entries on chromosome 22 (i.e. generated by the `Loci22` call). Each of these entries results in about 10-15 homologs being scanned by the `NA-Links` call, of which about 5 are non-human entries. So there are about 10,000 (nested) tuples in the result.

As would be expected, optimizations have a profound impact on the performance of this query, particularly on that for `Loci22` since it involves a relational server. When `Loci22` is implemented as written it involves three loops over the three GDB tables. The `locus` table would be retrieved in one access to GDB; each tuple in the `locus` table would then generate two additional accesses to GDB to find entries in `object_genbankeref` and `locus_cyto_location` that matched on the identifier fields. The resulting thousands and thousands of accesses to GDB not only takes forever but gets you kicked out of their system.

Migrating selections to GDB causes only those entries that are on chromosome 22 to be retrieved from `locus_cyto_location`; each of those entries then causes separate accesses to the `object_genbankeref` and `locus` tables. Total execution time of the query is roughly 602 seconds, with the response time to the initial result record being 270 seconds.

Migrating both selections and projections to GDB reduces the size of intermediate results and hence communication time from the server. Total execution time of the query becomes roughly 118 seconds, with the response time to the initial result record being 66 seconds.

When the join is also performed at the GDB server, the execution and response times become about 10-15 seconds. Most of this time is spent parsing the result.

4.4 Visualization

The graphical display of biological data is critical if users are to gain full value of the information. In fact, one of the primary reasons for the popularity of the ACeDB family of databases within the HGP is a well-thought out and comprehensive set of GUIs that include database schema and instance browsers, and integrated displays for genetic mapping, physical mapping, and sequence data among others [52]. However, as is the case with virtually all GUIs developed in the genomics community, the ACeDB GUIs are not portable across applications and are in fact closely wedded to the underlying database schemas. During the last several years, we have therefore created a set of GUIs written in Java, which were distilled from previous work done in Motif, Postscript [47], and tcl/Tk [48]. This GUI set, termed *BioWidget*⁸, was specifically designed with the goals of portability, reusability and rapid prototyping in mind. *BioWidgets* are used extensively in the online BioKleisli stereotypic queries previously described.

BioWidget encapsulates recurring themes in graphical objects and their behaviors, relieving the programmer of many tedious details. In addition, it achieves a common "look-and-feel" by way of features like a standard menubar and a common, context-sensitive help system. In addition to the general support and help widgets, the current *BioWidget* package includes: *map.class*, a system supporting the creation of various forms of abstract sequence schematics, genome maps and map objects on canvases; *multi-map.class*, an extension of *map.class* that simultaneously displays and coordinates multiple genetic and physical maps; *sequence.class*, a widget that creates a scrolling window of sequence data and again supports various domain-specific operations. A typical *BioWidget* applet, launched from Netscape in response to the query "Find primate genomic sequences with *alu* elements located inside a gene domain" is shown in Fig. 6. The applet displays the same data in the form of a sequence

⁸ The *BioWidget* package is now being adopted by a wide community of bioinformatics researchers and developers as part of a consortium effort. The master version is available for distribution from WWW site <http://agave.humgen.upenn.edu>.



Fig. 6. BioKleisli GUI implemented using BioWidgets

schematic created using the *map* widget and the annotated sequence (i.e. labeled sub-sequences) using the *sequence* widget. The two widgets communicate so that a feature or region selected in one is highlighted in the other. Textual information on selected features is extracted from the database and presented in yet another pop-up window.

5 Conclusions

Although in this paper CPL-Kleisli has been presented in the context of a particular application called BioKleisli, the system is in fact a general-purpose system which can query and combine information from a wide variety of heterogeneous data sources and application programs. The success of the system in doing this lies in the language CPL, the extensible architecture used in Kleisli, and the novel optimizations techniques. The complexity of types that CPL can handle (arbitrarily embedded records, variants, sets, lists and bags, with extensions to arrays as well as objects), and the simplicity yet power of the query language make it ideal for environments with standard (e.g. relational) as well as non-standard (e.g. scientific data formats) data sources. The optimization techniques not only extend well known techniques to these more complex types but are able to take advantage of the distributed environment by exploiting parallelism and exporting portions of the CPL query for execution at remote data sources.

As such, CPL-Kleisli contributes a solution to the problem of integrating heterogeneous data sources, a

topic that has been dealt with extensively in the computer science literature over the past 15 years [51, 49, 31, 39, 56, etc.]. The chief distinction between our approach and these is the complexity of data types that we model and query, the ability to transform between complex types, and the optimization techniques presented by Kleisli. Although the model in [4] 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 [43] which has a very simple data model and expresses types dynamically. However, when static type information is available it is extremely valuable for specifying and optimizing queries as illustrated with the stereotypic query in Sect. 4.

Data integration is also the focus of an ARPA-funded initiative called the “Intelligent Information Integration (I^3)”. This initiative is structured around the idea of *mediator* [57]. At a top level, mediators intelligently direct queries to the data sources that might contribute to an answer. A middle level deals with structuring and combining data as it is received from the lower layer, which ships queries out to the underlying data sources and translates data as it is returned. CPL-Kleisli contributes a solution to the lower two layers. We have not provided any assistance in determining where the relevant data is; the users must know this and express it in the CPL query. However, note that functions defining an integrated view of several underlying data sources could be used to provide some assistance, and this in fact is done within BioKleisli.

BioKleisli is one of the first serious attempts at creating an integrated digital library for biomedical researchers. In the introduction, we gave several examples of data warehouses and Netscape-based browsers that have been set up for biomedical researchers; however, they do not address the problem of querying and combining data from remote data sources and software packages. Using BioKleisli, we have therefore been able to perform almost all of the data retrieval tasks labeled as “impossible” in the 1993 DOE summit meeting report [21]; those that remain unanswered have not been attempted since some of the data that is needed is not currently available. From this perspective, as well as from the dramatic increase in web hits to our stereotypic queries, BioKleisli has been extremely successful.

Creating a digital library system for a particular area of interest – such as BioKleisli for the HGP – opens a number of opportunities that are not available when merely providing browsing access to arbitrary data sources over the internet. Since the locations of the major data sources are known, there is less need for web crawling queries to find potentially relevant data. Since the data management systems of relevant data sources are known (e.g. Sybase, Entrez, etc), generic drivers for those systems can be written; type information associated with the data can also be maintained and exploited during optimization. Since the general content of data sources is known (i.e. the data-types and their structure), queries can be written directly against the source schemas. Furthermore, since the ways in which the user community wants to interact with the data sources and visualize the results of queries is known, customized “user-views” of the data can be created and distributed as we have done for the stereotypic queries discussed in Sect. 4 using the BioWidget toolset.

Within the HGP, there is also some guarantee that the sequence data is “complete” since there are strong political and social incentives for scientists to submit sequence information to the major data sources.⁹ For example, many journals require submission of sequence information to GenBank prior to publication of an article. In this way, primary literature references can be maintained along with the actual sequence data; the GenBank accession number may also appear in the paper to enable database searches on related sequence information.

There are also efforts within the HGP community to provide semantic matching at the data level between the major data sources, for example, finding which protein sequence entries in PIR or loci in GDB correspond to nucleic acid sequence entries in GenBank. The formation of such links across data sources at the data level requires deep domain knowledge and the use of heuristic techniques to “guess” at the links. Surprisingly, considerable progress has been made in this area as evidenced by abstracts at the 1995 Meeting on the Interconnection of Molecular Biology Databases [34] reporting on a number of such linking tables. This goes hand-in-hand with the pragmatic Web-browser approaches being taken by

many bioinformatics systems developers [23, 6]. Within BioKleisli, these linking tables are also extremely useful for joining data across different data sources.

One of the biggest remaining challenges to forming the Biomatrix is, in our opinion, dealing with the complexity of schemas when writing ad-hoc queries across multiple databases, i.e. providing support for the “mediator” layer. In implementing the impossible queries as well as several others given to us by the community, we found that it was sometimes extremely difficult to determine exactly which data sources contribute best to the answer. We believe that this problem is most likely to be solved by domain experts using good schema documentation tools (such as those in [18]) and graphical front-ends for query formulating (such as those in [50]). However, the problem is far from solved with these approaches.

Acknowledgements. We would like to thank Peter Buneman for his help, suggestions and examples, and Kyle Hart and Jonathan Crabtree for their implementation work on Kleisli. Serge Abiteboul also provided many helpful comments on an earlier version of this paper.

References

1. S. Abiteboul, P. Fischer, and H.-J. Schek, editors. *LNCS 361: Nested Relations and Complex Objects in Databases*. Springer-Verlag, 1989.
2. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.
3. Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *The VLDB Journal*, 4:727–794, 1995.
4. Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.
5. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
6. Y. Akiyama, S. Goto, I. Uchiyama, and M. Kanehisa. Web-DBGET: an integrated database retrieval system which provides hyper-links among related database entries, 1995. In abstracts of participants of the 1995 Meeting on Interconnecting Molecular Biology Databases. Available at url <http://www.genome.ad.jp/dbget/dbget.html>.
7. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.
8. Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.
9. O. P. Buneman and A. Ohori. A type system that reconciles classes and extents. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 191–202. Morgan Kaufmann, August 1991.

⁹ Although the data is relatively complete it is not without error. There are major efforts to annotate and clean up sequence data.

10. Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.
11. Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.
12. Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.
13. Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.
14. L. Cardelli. A semantics for multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.
15. Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, Minnesota, May 1994.
16. R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1996.
17. I-Min A. Chen and Victor M. Markowitz. The object-protocol model, draft 2. Technical Report LBL-32738, Lawrence Berkeley Laboratory, Berkeley, California, 1994.
18. I.A. Chen and V.M. Markowitz. Modeling Scientific Experiments with an Object Data Model. In *Proc. 11th Int. Conference on Data Engineering*, pages 391–400, 1995.
19. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.
20. Committee on Models for Biomedical Research. *Models for Biomedical Research: A New Perspective*. National Academy Press, Washington, D.C., 1985.
21. Department of Energy. *DOE Informatics Summit Meeting Report*, April 1993. Available via gopher at gopher.gdb.org.
22. O. Deux. The story of O₂. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.
23. Thure Etzold and Patrick Argos. SRS: An indexing and retrieval tool for flat file data libraries. *Computer Applications of Biosciences*, 9:49–57, 1993.
24. Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–58, San Jose, California, May 1995.
25. Nathan Goodman, Mary-Pat Reeve, and Lincoln Stein. The design of mapbase: An object oriented database for genome mapping. Technical report, Whitehead Institute for Biomedical Research, One Kendall Square, Cambridge, MA, 1992.
26. C. Hara and L. Popa. Querying Shore using CPL, August 1996. Manuscript available from chara@saul.cis.upenn.edu.
27. L. A. Jategaonkar and J. C. Mitchell. Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, 19:127–165, 1993.
28. Won Kim. A new way to compute the product and join of relations. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 179–187, 1980.
29. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates, August 1996. Manuscript available from val@saul.cis.upenn.edu.
30. Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 228–239, Montreal, Canada, June 1996.
31. W. Litwin and A. Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(3):10–18, December 1986.
32. David Maier and Bennet Vance. A call to order. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 1–16, Washington, D. C., May 1993.
33. V. Markowitz, editor. *Journal of Computational Biology*, volume 2. Mary Ann Liebert, Inc, Winter 1995.
34. Meeting on Interconnection of Molecular Biology Databases. Abstract of participants, July 1995. Available at url <http://www.ai.sri.com/~pkarp/mimbd/95/abstracts.html>.
35. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.
36. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
37. M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of Conference on Very Large Databases*, pages 468–478, 1988.
38. National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD. *NCBI ASN.1 Specification*, 1992. Revision 2.0.
39. S. Navathe, S.Gala, S. Geum, A. Kamath, A. Krishnaswamy, A. Savasere, and W. Whang. A federated architecture for heterogeneous information systems. In *Workshop on Heterogeneous Databases*. National Science Foundation, December 1989.
40. NCBI. ENTREZ: Sequences users's guide. Technical Report Release 1.0, National Center For Biotechnology Information, National Library of Medicine, Bethesda, MD, 1992.
41. A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.
42. Overbeek et al. GenoBase Database Gateway, July 1995. Available at url <http://specter.dcrct.nih.gov:8004>.
43. Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.
44. P. Pearson, N. Matheson, N Flescher, and R. J. Robbins. The GDB human genome data base anno 1992. *Nucleic Acids Research*, 20:2201–2206, 1992.
45. R. Rew, G. Davis, S. Emmerson, and H. Davies. NetCDF User's Guide: An Interface for Data Access. Available at www.unidata.ucar.edu/packages/netcdf/guide.toc.html.
46. Otto Ritter. The IGD approach to the interconnection of genomic databases. In *Abstracts of 1st Meeting on Integration of Molecular Biology Databases*, Stanford University, Stanford CA, August 1994.
47. D.B. Searls. Doing sequence analysis with your printer. *CABIOS*, 9(4):421–426, 1993.
48. D.B. Searls. *bioTk*: componentry for genome informatics graphical user interfaces. *Gene*, 163(2):GC1–16, 1995. (appeared electronically in Gene-COMBIS).
49. J. Smith, P. Bernstein, U. Dayal, N. Goodman, T. Landers, K. Lin, and E. Wong. Multibase — Integrating heterogeneous distributed database systems. In *Proceedings of AFIPS*, pages 487–499, 1981.
50. Wang Chiew Tan, Ke Wang, and Limsoon Wong. QUICK: A graphical user interface to multiple databases, May 1996. Available at <http://sdmc.iss.nus.sg/kleisli/limsoonpapers.html>.
51. M. Templeton, D. Brill, S. Dao, E. Lund, P. Ward, A. Chen, and R. MacGregor. Mermaid — a front-end to distributed heterogeneous databases. *Proceedings of the IEEE*, 75(5):695–708, May 1987.

52. Jean Thierry-Mieg and Richard Durbin. *acedb — A C.elegans Database: Syntactic Definitions for the ACEDB Data Base Manager*, 1992.
53. P. W. Trinder. Comprehensions, a query notation for DB-PLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nafplion, Greece*, pages 49–62. Morgan Kaufmann, August 1991.
54. P. W. Trinder and P. L. Wadler. Improving list comprehension database queries. In *Proceedings of TENCON'89, Bombay, India*, pages 186–192, November 1989.
55. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
56. S. Widjojo, D. S. Wile, and R. Hull. Worldbase: A new approach to sharing distributed information. Technical report, USC/Information Sciences Institute, February 1990.
57. Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.
58. L. Wong. *The Collection Programming Language Reference Manual*. Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, October 1995. Available at <http://sdmc.iss.nus.sg/kleisli/psZ/cpl-defn.ps>.
59. L. Wong. *The Kleisli Query System Reference Manual*. Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, October 1995. Available at <http://sdmc.iss.nus.sg/kleisli/psZ/kleisli-manual.ps>.
60. L. Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52, 1996. To appear.
61. Limsoon Wong. *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.
62. Limsoon Wong. *The Kleisli/CPL Extensible Query Optimizer Programmer Guide*. Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, November 1995. Available at <http://sdmc.iss.nus.sg/kleisli/psZ/cplopt.ps>.

This article was processed by the author using the L^AT_EX style file *cljour2* from Springer-Verlag.