The Kleisli Query System as a Backbone for Bioinformatics Data Integration and Analysis

Jing Chen
geneticXchange Inc
chenjing@geneticxchange.com

Su-Yun Chung
geneticXchange Inc
suchung@sdsc.edu

Limsoon Wong
Laboratories for Information Technology
limsoon@lit.org.sg

April 16, 2002

Abstract

Kleisli is a data transformation and integration system that can be used for any application where the data is typed. It has proven to be especially useful for bioinformatics applications. It extends the conventional flat relational data model supported by the query language SQL to a complex object data model supported by a "nested-relationalized" version of SQL called sSQL. It also opens up the closed nature of commercial relational data management systems to an easily extensible system that performs complex transformations on autonomous data sources that are heterogeneous and geographically dispersed. This paper comments on various aspects of Kleisli such as its approach, data representation, query capacity, data sources, optimizations, user interfaces, etc. This paper also presents some example bioinformatics applications of Kleisli.

1 Introduction

Biological data is characterized by a wide range of data types from the plain text of laboratory records and literatures, nucleic acid and amino acid sequences, three-dimensional structures of molecules, high-resolution images of cells and tissues, diagrams of biochemical pathways and regulatory networks, to various experimental outputs from technologies as diverse as microarrays, gels, and mass spectrometry. These data are stored in a large number of databases across the Internet. In addition to online interface for querying and searching the underlying repository data, many database sites also provide specific computational tools or programs for analysis of data. We use the term data sources to loosely refer to both databases and computational analysis tools.

Until recently, data sources were set up as autonomous sites by individual institutions or research laboratories. Data sources vary considerably in contents, access methods, capacity, query processing, and services. The major difficulty is that the data elements in various public and private data sources are stored in extremely heterogeneous formats and database management systems that are often ad hoc, application-specific, or vendor-specific. For example, the scientific literature, patents, images, and other free-text documents are commonly stored in unstructured formats like plain text files, HTML files, and binary files. Genomic, microarray gene expression, or proteomic data are routinely stored in Excel spread sheets, semi-structured XML, or structured relational databases like Oracle, Sybase,DB2, Informix, etc. The National Center for Biotechnology Information in Bethesda (NCBI), which is the largest repository for genetic information, supplies GenBank reports, GenPept Reports, etc. in HTML format with an underlying highly nested data model based on ASN.1 [21]. The computational analysis tools or applications suffer from a similar scenario: they require specific input and output data formats. The output of one program is not immediately compatible to the input requirement of other programs. For example, the most popular BLAST database search tool require a specific format called FASTA for sequence input.

In addition to data format variations, both the data content and data schemas of these databases are constantly changing in response to rapid advances in research and technology. As the amount of data and databases continue to grow in the Internet, it generates another bottleneck in information integration at the semantic level. There is a general lack of standards in controlled vocabulary for consistent naming of biomedical terms, functions, and processes within and between databases. In naming genes and proteins alone, there is much confusion. For example, a simple transcription factor, the CCAAT/enhancer-binding protein beta, is referred to by more than a dozen of names in the public databases as CEBPB, CRP2, IL6DPB, etc.

For research and discovery, the biologists needs to access to up-to-date data and best-of-breed computational tools for data analyses. To achieve this goal, the ability to query across multiple data sources is not enough. It also demands means to transform and transport data through various computational steps seamlessly. For example, to investigate the structure and function of a new protein, the users must integrate information derived from sequence, structure, protein domain prediction, and literature data sources. If these steps have to be carried out manually that requires some level of programming work (such as writing Perl scripts) to prepared the data sets between the output of one step to the input of the next step, the process would be very inefficient and slow.

In short, many bioinformatics problems require access to data sources that are large, highly heterogeneous and complex, constantly evolving, and geographically dispersed. Solutions to these problems usually involve many steps and require information to be passed smoothly and usually transformed between the steps. The Kleisli system is designed to handle these requirements directly by providing a high-level query language, sSQL, that can be used to express complicated transformations across multiple data sources in a clear and simple way.[1]

The design and implementation of the Kleisli system are heavily influenced by functional programming research, as well as database query language research. Its high-level query language sSQL can be considered as a functional programming language that has a built-in notion of "bulk" data types suitable for database programming and has many built-in operations required for modern bioinformatics. Kleisli is itself implemented on top of the functional programming language Standard ML of New Jersey (SML). Even the data format that Kleisli uses to exchange information with the external world is derived from ideas in type inference.

In this chapter, we provide a description of the Kleisli system and a discussion on various aspects of the system such as data representation, query capability, optimizations, and user interfaces. The materials are organized as follows. Section 2 introduces Kleisli with a well-known example. Section 3 presents an overview of the Kleisli system. Section 4 discusses the data model, data representation, and exchange format of Kleisli. Section 5 gives more example queries in Kleisli and comments on the expressive power of its core query language. Section 6 illustrates Kleisli's ability to use flat relational databases to transparently store complex objects. Section 7 lists the kind of data sources supported by the Kleisli system and shows the ease of implementing wrappers for Kleisli. Section 8 gives an overview of the various types of optimizations performed by the Kleisli query optimizer. Section 9 describes both the ODBC- or JDBC-like programming interface to Kleisli in Perl and Java, as well as its Discovery Builder graphical user interface. Section 10 contains a brief comparison of the Kleisli system to other well-known proposals for bioinformatics data integration.

2 Motivating Example

Before we discuss the guts of the Kleisli system, let us first repeat below the very first bioinformatics data integration problem solved using Kleisli. The query was implemented in Kleisli in 1994 [12] and solved one of the so-called "impossible" queries of a US Department of Energy Bioinformatics Summit Report published in 1993.[2]

[1] Earlier versions of the Kleisli system supported only a query language based on comprehension syntax called CPL [30]. Now, both CPL and sSQL are available.

[2] The report is available at `www.gdb.org/Dan/DOE/whitepaper/contents.html`.

Example 2.1 *The query was to find for each gene located on a particular cytogenetic band of a particular human chromosome as many of its non-human homologs as possible. Basically, the query means that for each gene in a particular position in the human genome, find DNA sequences from non-human organisms that are similar to it.*

In 1994, the main database containing cytogenetic band information was the GDB [22], which was a Sybase relational database. In order to find homologs, the actual DNA sequences were needed and the ability to compare them was also needed. Unfortunately, that database did not keep actual DNA sequences. The actual DNA sequences were kept in another database called GenBank [9]. At the time, access to GenBank was provided through the ASN.1 version of Entrez [24], which was an extremely complicated retrieval system. Entrez also kept precomputed homologs of GenBank sequences.

So this query needed the integration of GDB (a relational database located in Baltimore) and Entrez (a non-relational "database" located in Bethesda). The query first extracted the names of genes on the desired cytogenetic band from GDB, and then accessed Entrez for homologs of these genes. Finally, these homologs were filtered to retain the non-human ones. This query was considered "impossible" as there was at that time no system that could work across the bioinformatics sources involved due to their heterogeneity, complexity, and geographical locations. Given the complexity of this query, the sSQL solution below is remarkably short.

```
sybase-add (name: "gdb", ...);
create view locus from locus_cyto_location using gdb;
create view eref from object_genbank_eref using gdb;
select accn: g.genbank_ref, nonhuman-homologs: H
from
  locus c, eref g,
  {select  u
   from na-get-homolog-summary(g.genbank_ref) u
   where not(u.title like "%Human%") and not(u.title like "%H.sapien%")} H
where
  c.chrom_num = "22" and g.object_id = c.locus_id and not (H = { });
```

The first three lines connect to GDB and map two tables in GDB to Kleisli. After that, these two tables could be referenced within Kleisli as if they were two locally defined sets, `locus` *and* `eref`*. The next few lines extract from these tables the accession numbers of genes on Chromosome 22, use the Entrez function* `na-get-homolog-summary` *to obtain their homologs, and filter these homologs for non-human ones. Notice that in the* `from`*-part of the outer* `select`*-construct, we have* {`select u ...  }` `H`*. This means that* `H` *is the entire set returned by* `select u ...`*, and thus allowing us to manipulate and return all the non-human homologs as a single set* `H`*.*

Besides the obvious smoothness of integration of the two data sources, this query is also remarkably efficient. On the surface, it seems to fetch the `locus` *table in its entirety once and the* `eref` *table in its entirety n times from GDB, as a naive evaluation of the comprehension would be two nested loops iterating over these two tables. Fortunately, in reality, the Kleisli optimizer is able to migrate the join, selection, and projections on these two tables into a single efficient access to GDB using the optimizing rules from a later section. Furthermore, the accesses to Entrez are also automatically made concurrent.* □

Since the query above, Kleisli and its components have been used in a number of bioinformatics projects such as GAIA at the University of Pennsylvania (www.cis.upenn.edu/gaia2), TAMBIS at the University of Manchester [4], and FIMM at Kent Ridge Digital Labs [23]. It has also been used in constructing databases in pharmaceutical/biotechnology companies such as SmithKline Beecham, Schering-Plough, GlaxoWellcome, Genomics Collaborative, Signature Biosciences, etc. Kleisli is also the backbone of GeneticXchange Inc. (www.geneticxchange.com).

3 Approach

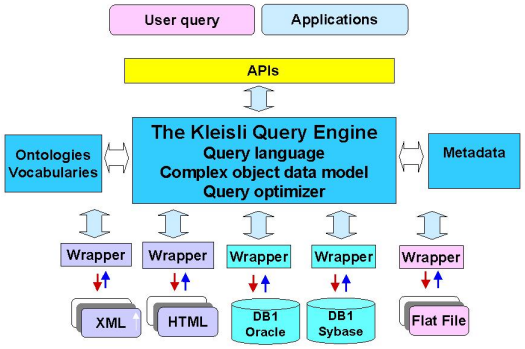

Figure 1: Kleisli, positioned as a mediator

The approach taken by the Kleisli system is illustrated by the diagram in Figure 1. It is positioned as a mediator system encompassing a complex object data model, a high-level query language, and a powerful query optimizer. It runs on top of a large number of light-weight wrappers for accessing various data sources. There are also a number of application programming interfaces that allow Kleisli to be accessed in a ODBC- or JDBC-like fashion in various programming languages for a various applications.

The Kleisli system is extensible in several ways. It can be used to support several different high-level query languages by replacing its high-level query language module. Currently, Kleisli supports a "comprehension syntax"-based language called CPL [7, 27, 30] and a "nested relationalized" version of SQL called sSQL. We use sSQL throughout this chapter. The Kleisli system can also be used to support many different types of external data sources by adding new wrappers, which forward Kleisli's requests to these sources and translate their replies into Kleisli's exchange format. These wrappers are light weight and new wrappers are generally easy to develop and insert into the Kleisli system. The optimizer of the Kleisli system can also be customized by different rules and strategies.

When a query is submitted to Kleisli, it is first processed by the high-level query language module which translates it into an equivalent expression in the abstract calculus NRC. NRC is based on that described in [8], and is chosen as the internal query representation because it is easy to manipulate and amenable to machine analysis. The NRC expression is then analyzed to infer the most general valid type for the expression, and is passed to the query optimizer. Once optimized, the NRC expression is then compiled into calls to a library of routines for complex objects underlying the complex object data model. The resulting compiled code is then executed, accessing drivers and external primitives as needed through pipes or shared memory.

We consider each of these components in further detail in the next several sections.

4 Data Model and Representation

The data model, data representation, and data exchange format of the Kleisli system are presented in this section. The data model underlying the Kleisli system is a complex object type system that goes beyond the "sets of records" or "flat relations" type system of relational databases [11]. It allows arbitrarily nested records, sets, lists, bags, and variants. A variant is also called a tagged union type, and represents a type that is "either this or that". The collection or "bulk" types–sets, bags, and lists–are homogeneous. In order to mix objects of different types in a set, bag, or list, it is necessary to inject these objects into a variant type.

In a relational database, the sole bulk data type is the set. Having only one bulk data type presents at least two problems in real life applications. Firstly, the particular bulk data type may not be a natural model of real data. Secondly, the particular bulk data type may not be an efficient model of real data. For example, if we are restricted to the flat relational data model, the GenPept report in Example 4.1 below must necessarily be split into many separate tables in order to be losslessly stored in a relational database. The resulting multi-table representation of the GenPept report is conceptually unnatural and operationally inefficient. A person querying the resulting data must pay the mental overhead of understanding both the original GenPept report and its badly-fragmented multi-table representation. He may also have to pay the performance overhead of having to re-assemble the original GenPept report from its fragmented multi-table representation to answer queries.

Example 4.1 *The GenPept report is the format chosen by the US National Center for Biotechnology Information to present amino acid sequence information. While an amino acid sequence is a string of letters, certain regions and positions of the string are of special biological interest, such as binding sites, domains, and so on. The feature table of a GenPept report is the part of the GenPept report that documents the positions of these regions of special biological interest, as well as annotations or comments on these regions. The following type represents the feature table of a GenPept report from Entrez [24].*

```
(#uid:num, #title:string,
  #accession:string, #feature:{(
     #name:string, #start:num, #end:num,
     #anno:[(#anno_name:string, #descr:string)])})
```

It is an interesting type because one of its fields (`#feature`*) is a set of records, one of whose fields (*`#anno`*) is in turn a list of records. More precisely, it is a record with four fields* `#uid`, `#title`, `#accession`, *and* `#feature`. *The first three of these store values of types* `num`, `string`, *and* `string` *respectively. The* `#uid` *field uniquely identifies the GenPept report. The* `#feature` *field is a set of records, which together form the feature table of the corresponding GenPept report. Each of these records has four fields* `#name`, `#start`, `#end`, *and* `#anno`. *The first three of these have types* `string`, `num`, *and* `num` *respectively. They represent the name, start position, and end position of a particular feature in the feature table. The* `#anno` *field is a list of records. Each of these records has two fields* `#anno_name` *and* `#descr`, *both of type* `string`. *These records together represent all annotations on the corresponding feature.* □

In general, the types are freely formed by the syntax:

$$t ::= \texttt{num} \mid \texttt{string} \mid \texttt{bool} \mid \{t\} \mid \{|\,t\,|\} \mid [t] \mid (l_1 : t_1, ..., l_n : t_n) \mid \langle l_1 : t_1, ..., l_n : t_n \rangle$$

Here `num`, `string`, and `bool` are the base types. The other types are constructors and build new types from existing types. The types $\{t\}$, $\{|\,t\,|\}$, and $[t]$ respectively construct set, bag, and list types from type t. The type $(l_1 : t_1, ..., l_n : t_n)$ constructs record types from types t_1, ..., t_n. The type $\langle l_1 : t_1, ..., l_n : t_n \rangle$ constructs variant types from types t_1, ..., t_n. The flat relations of relational databases are basically sets of records, where

each field of the records is a base type; in other words, relational databases have no bags, no lists, no variants, no nested sets, and no nested records. Values of these types can be explicitly represented and exchanged as follows, assuming the e's are values of appropriate types: $(l_1 : e_1, ..., l_n : e_n)$ for records; $<l : e>$ for variants; $\{e_1, ..., e_n\}$ for sets; $\{|e_1, ..., e_n|\}$ for bags; and $[e_1, ..., e_n]$ for lists.

Example 4.2 *The feature table of GenPept report 131470, a tyrosine phosphatase 1C sequence, is shown below.*

```
(#uid:131470, #accession:"131470",
  #title:"... (PTP-1C)...", #feature:{(
   #name:"source", #start:0, #end:594, #anno:[
    (#anno_name:"organism", #descr:"Mus musculus"),
    (#anno_name:"db_xref", #descr:"taxon:10090")]),
   ...})
```

The particular feature displayed above goes from amino acid 0 to amino acid 594, which is actually the entire sequence, and has two annotations: The first annotation indicates that this amino acid sequence is derived from mouse DNA sequence. The second is a cross reference to the US National Center for Biotechnology Information taxonomy database. □

The schemas and structures of all popular bioinformatics databases, flat files, and softwares are easily mapped into this data model. At the high end of data structure complexity are Entrez [24] and ACEDB [28], which contain deeply nested mixtures of sets, bags, lists, records, and variants. At the low end of data structure complexity are the relational database systems [11] such as Sybase and Oracle, which contain flat sets of records. Currently, Kleisli gives access to over sixty of these and other bioinformatics sources. The reason for this ease of mapping bioinformatics sources to Kleisli's data model is that they are all inherently composed of combinations of sets, bags, lists, records, and variants. We can directly and naturally map sets to sets, bags to bags, lists to lists, records to records, and variants to variants into Kleisli's data model, without having to make any (type) declaration before hand.

The last point above deserves further consideration. In a dynamic heterogeneous environment such as that of bioinformatics, many different database and software systems are used. They often do not have anything that can be thought of as an explicit database schema. Further compounding the problem is that research biologists demand flexible access and queries in ad-hoc combinations. Thus, a query system that aims to be a general integration mechanism in such an environment must satisfy four conditions. First, it must not count on the availability of schemas. It must be able to compile any query submitted based solely on the structure of that query. Second, it must have a data model that the external database and software systems can easily translate to, without doing a lot of type declarations. Third, it must shield existing queries from evolution of the external sources as much as possible. For example, an extra field appearing in an external database table must not necessitate the recompilation or rewriting of existing queries over that data source. Fourth, it must have a data exchange format that is straightforward to use, so that it does not demand too much programming effort or contortion to capture the variety of structures of output from external databases and softwares.

Three of these requirements are addressed by features of sSQL's type system. sSQL has polymorphic record types that allow, for example,

```
create function get-rich-guys (R) as
select x.name from R x where x.salary > 1000;
```

which defines a function that returns names of people in `R` earning more than a thousand dollars. This function is applicable to any `R` that has at least the `name` and the `salary` fields, thus allowing the input source some freedom to evolve.

In addition, sSQL does not require any type to be declared at all. The type and meaning of any sSQL program can always be completely inferred from its structure without the use of any schema or type declaration. This makes it possible to logically plug in any data source without doing any form of schema declaration, at a small acceptable risk of run-time errors if the inferred type and the actual structure are not compatible. This is an important feature because most of our data sources do not have explicit schemas, while a few have extremely large schemas that take many pages to write down—for example, the ASN.1 schema of Entrez [21]—making it impractical to have any form of declaration.

We now come to the fourth requirement. A data exchange format is an agreement on how to lay out data in a data stream or message when the data is exchanged between two systems. In our case, it is the format for exchanging data between Kleisli and all the bioinformatics sources. The data exchange format of Kleisli corresponds one-to-one to Kleisli's data model. It provides for records, variants, sets, bags, and lists; and it allows these data types to be freely composed. In fact, the data exchange format completely adopts the syntax of data representation described earlier and illustrated in Example 4.2. This representation has the interesting property that it has no ambiguity. For instance, whenever we see a { we know that we have a set, whenever we see a (we know that we have a record, etc. In short, this data exchange format is self describing. See [31] for more detail on the data exchange format of Kleisli.

A self-describing exchange format is one in which there is no need to define in advance the structure of the objects being exchanged. That is, there is no fixed schema and no type declaration. In a sense, each object being exchanged carries its own description. A self-describing format has the important property that, no matter how complex the object being exchanged is, it can be easily parsed and reconstructed without any schema information. To understand this advantage, one should look at the ISO ASN.1 standard [18] open systems interconnection. It is not easy to exchange ASN.1 objects because before we can parse any ASN.1 object, we need to parse the schema that describes its structure first—making it necessary to write two complicated parsers instead of one simple parser.

5 Query Capability

We now come to sSQL, the primary query language of Kleisli used in this chapter. It is based on the *de facto* commercial database query language SQL, except for extensions that we have made to cater for the nested relational model and for the federated heterogeneous data sources. Rather than giving the complete syntax, we illustrate sSQL by a few examples on a set of feature tables `DB`.

Example 5.1 *The query below extracts the titles and features of those elements of* `DB` *whose titles contain* `tyrosine` *as a substring.*

```
create function get-title-from-featureTable (DB) as
select title: x.title, feature: x.feature
from DB x where x.title like "%tyrosine%";
```

□

This query is a simple project-select query. A project-select query is a query that operates on one (flat) relation or set. Thus the transformation that such a query can perform is limited to selecting some elements of the relation and extracting or projecting some fields from these elements. Except for the fact that the source data and the result may not be in first normal form, these queries can be expressed in a relational query language. However, sSQL can perform more complex restructurings such as nesting and unnesting not found in SQL, as shown in the following examples.

Example 5.2 *The following query flattens* DB *completely.*[3]

```
create function flatten-featureTable (DB) as
select
  title:x.title, feature:f.name, start:f.start, end:f.end,
  anno-name:a.anno_name, anno-descr:a.descr
from DB x, x.feature f, f.anno.l2s a;
```

□

Example 5.3 *This query demonstrates how to do nesting in sSQL. Notice that the* entries *field is a complex object having the same type as* DB.

```
create function nest-featureTable-by-organism (DB) as
select
  organism: z,
  entries: (select distinct x
            from DB x, x.feature f, f.anno a where a.anno_name = "organism" and a.descr = z)
from (select distinct y.anno-descr from DB.flatten-featureTable y where y.anno-name ="organism") z;
```

□

We can now proceed to the more ambitious example of an *in silico* discovery kit. Such a kit prescribes experimental steps carried out in computers very much like the experimental protocol carried out in wet laboratories for specific scientific investigation. From the perspective of Kleisli, an *in silico* discovery kit is just a script written in sSQL and performs a defined information integration task very similar to an integrated electronic circuit. It takes an input data set and parameters from the user, executes and integrates the necessary computational steps of database queries and applications of analysis programs or algorithms, and outputs a set of results for specific scientific inquiry.

Example 5.4 *The simple in silico discovery kit in Figure 2 demonstrates how to use an available ontology data source to get around the problem of inconsistent naming in genes and proteins, and to integrate information across multiple data sources. It is implemented in the sSQL script below.*[4] *With the user input of a gene name* G*, the ISDK performs the following task: First, it retrieves a list of aliases for* G *from the Gene Nomenclature database provided by the Human Genome Organization (HUGO). Then it retrieves information for diseases associated with this particular protein in the Online Mendelian Inheritance of Man Database (OMIM), and finally it retrieves all relevant references from MEDLINE.*

```
create function get-info-by-genename (G) as
Select
  hugo: w, omim: y, pmid1-abstract: z,
  num-medline-entries: list-sum(lselect ml-get-count-general(n) from x.Aliases.s2l n)
from
```

[3] `l2s` is a function that converts a list into a set.

[4] `s2l` is a function that converts a set into a list. `list-sum` is a function to sum a list of numbers. `ml-get-count-general` is a function that accesses the MEDLINE database in Bethesda and computes the number of MEDLINE reports matching a given keyword. `ml-get-abstract-by-uid` is a function that accesses MEDLINE for report given a unique identifier. `webomim-get-id` is a function that accesses the OMIM database in Bethesda to obtain unique identifiers of OMIM reports matching a keyword. `webomim-get-detail` is a function that accesses OMIM for report given a unique identifier. `hugo-get-by-symbol` is a function that accesses the HUGO database and return HUGO reports matching a given gene name.

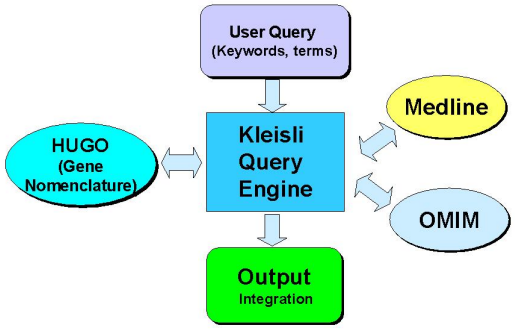

Figure 2: An "*in silico* discovery kit" that uses an available ontology data source to get around the problem of inconsistent naming in genes and proteins, and integrates information across multiple data sources.

```
  hugo-get-by-symbol(G) w,
  webomim-get-id(searchtime:0, maxhits:0, searchfields:{}, searchterms:G) x,
  webomim-get-detail(x.uid) y,
  ml-get-abstract-by-uid(w.PMID1) z
where
  x.title like ("%" ^ G ^ "%");
```

For instance, this query `get-info-by-genename` *can be invoked with the transcription factor* `CEBPB` *as input to obtain the following result.*

```
{(#hugo: (#HGNC: "1834",
   #Symbol: "CEBPB", #PMID1: "1535333", ...
   #Name: "CCAAT/enhancer binding protein (C/EBP), beta",
   #Aliases: {"LAP", "CRP2", "NFIL6", "IL6DBP", "TCF5"}),
  #omim: (#uid: 189965, #gene_map_locus: "20q13.1", #allelic_variants: {}, ...),
  #pmid1-abstract: (#muid: 1535333,
   #authors: "Szpirer C...", #address: "Departement de Biologie ...",
   #title: "Chromosomal localization in man and rat of the genes encoding ...",
   #abstract: "By means of somatic cell hybrids segregating either human...",
   #journal: "Genomics 1992 Jun; 13(2):292-300"),
  #num-medline-entries: 1936)}
```

□

Such queries fulfil many of the requirements for efficient *in silico* discovery processes: (1) their modular nature gives scientists the flexibility to select and combine specific queries for specific research project; (2) they can be executed automatically by Kleisli in batch mode and can handle large data volume; (3) their scripts are reusable to perform repetitive tasks and can be shared among scientific collaborators; (4) they form a base set of templates that can be readily modified and refined to meet different specifications and to make new queries; and (5) new databases and new computational tools can be readily incorporated to existing scripts.

While the syntactic basis for sSQL is SQL, its theoretical inspiration came from [5] where structural recursion was presented as a query language. However, structural recursion has two difficulties. The first is that not every syntactically correct structural recursion program is logically well defined [6]. The second is that structural recursion has too much expressive power because it can express queries that require exponential time and space.

In the context of databases, which are typically very large, programs (queries) are usually restricted to those which are "practical" in the sense that they are in a low complexity class such as LOGSPACE, PTIME, or TC^0. In fact, one may even want to prevent any query that has worse than $O(n \cdot \log n)$ complexity, unless one is confident that the query optimizer has a high probability of optimizing the query to no more than $O(n \cdot \log n)$ complexity. Database query languages such as SQL are therefore designed in such a way that joins are easily recognized, since joins are the only operations in a typical database query language that require $O(n^2)$ complexity if evaluated naively.

Thus Tannen and Buneman suggested a natural restriction on structural recursion to reduce its expressive power and to guarantee its well-definedness. Their restriction cuts structural recursion down to homomorphisms on the commutative idempotent monoid of sets, revealing a telling correspondence to monads [27]. A nested relational calculus, which is denoted here by $\mathcal{NRC}$, was then designed around this restriction [8]. $\mathcal{NRC}$ is essentially the simply-typed lambda calculus extended by a construct for building records, a construct for decomposing records by field selection, a construct for building sets, a construct for decomposing sets by means of the restriction on structural recursion. Specifically, the construct for decomposing sets is $\bigcup\{e_1 \mid x \in e_2\}$, which forms a set by taking the big union of $e_1[o/x]$ over each o in the set e_2.

The expressive power of $\mathcal{NRC}$ and its extensions are studied in [25, 13, 19, 8, 26]. Specifically, we know that the $\mathcal{NRC}$ core has exactly the same power as all the standard nested relational calculi and that when restricted to flat tables as input-output it has exactly the same power as the relational calculus. In the presence of arithmetics and a summation operator, when restricted to flat tables as input-output, it has exactly the power of entry-level SQL. Furthermore, it captures standard nested relational queries in a high-level manner that is easy for automated optimizer analysis. It is also easy to translate a more user-friendly surface syntax such as the comprehension syntax or the SQL select-from-where syntax into this core, while allowing for full-fledged recursion and other operators to be imported easily as needed into the system.

6 Warehousing Capability

Besides the ability to query, assemble, and transform data from remote heterogeneous sources, it is also important to be able to conveniently warehouse the data locally. The reasons to create local warehouses are several: (1) it increases efficiency; (2) it increases availabilty; (3) it reduces risk of unintended "denial of service" attacks on the original sources; and (4) it allows more careful data cleansing that cannot be done on the fly. The warehouse should be efficient to query and easy to update. Equally important in the biology arena is that the warehouse should model the data in a conceptually natural form. Although a relational database system is efficient for querying and easy to update, its native data model of flat tables forces us to unnaturally and unnecessarily fragment our data in order to fit our data into third normal form.

Kleisli does not have its own native database management system. Instead, Kleisli has the ability to turn many kinds of database systems into an updatable store conforming to its complex object data model. In particular, Kleisli can use flat relational database management systems such as Sybase, Oracle, MySQL, etc. to be its updatable complex object store. It can even use all of these systems simultaneously! We illustrate this power of Kleisli using the example of GenPept reports.

Example 6.1 *Create a warehouse of GenPept reports.*[5] *Initialize it to reports on protein tyrosine phosphatases.*

[5] Kleisli provides several functions to access GenPept reports remotely from Entrez [24]. One of them is `aa-get-seqfeat-general`,

```
! connect to our Oracle database system
oracle-cplobj-add (name: "db", ...);
! create a table to store GenPept reports
create table genpept(uid: "NUMBER", detail: "LONG") using db;
! initialize it with PTP data
select (uid: x.uid, detail: x) into genpept from aa-get-seqfeat-general("PTP") x using db;
! index the uid field for fast access
db-mkindex (table: "genpept", index: "genpeptindex", schema: "uid");
! let's use it now to see the title of report 131470
create view GenPept from genpept using db;
select x.detail.title from GenPept x where x.uid = 131470;
```

In this example, a table `genpept` *is created in our Oracle database system. This table has two columns,* `uid` *for recording the unique identifier and* `detail` *for recording the GenPept report. A LONG data type is used for the* `detail` *column of this table. However, recall from Example 4.2 that each GenPept report is a highly nested complex objects. There is therefore a "mismatch" between LONG (which is essentially a big uninterpreted string) and the complex structure of a GenPept report. This mismatch is resolved by the Kleisli system which automatically performs the appropriate encoding and decoding. Thus, as far as the Kleisli user is concerned,* `x.detail` *has the type of GenPept report as given in Example 4.1. So he can ask for the title of a report as straightforwardly as* `x.detail.title`*. Note that encoding and decoding are performed to mapped the complex object transparently into the space provided in the* `detail` *column; that is, the Kleisli system does not fragment the complex object to force it into third normal form.* □

7 Data Sources

The standard version of the Kleisli system marketed by geneticXchange Inc (`www.geneticxchange.com`) supports over 60 types of data sources. These include the categories below.

- Relational database management systems: All popular relational database management systems are supported, such as Oracle, Sybase, DB2, Informix, and MySQL. The support for these systems is quite sophisticated. For example, in the previous section, we have just seen that the Kleisli system can transparently turn these flat database systems into efficient complex object stores. In a later section, we also show that the Kleisli system is able to perform a significant amount of query optimizations involving these systems,
- Bioinformatics analysis packages: Most popular analysis packages for analysis of protein sequences and other biological data are supported. These packages include both web-based and/or locally-installed versions of WU-BLAST [1], Gapped BLAST [2], FASTA, ClustalW, HMMER, BLOCKS, Pfscan, nnPredict, PSORT, and many others.
- Biological databases: Many popular data sources of biological information are also supported by the Kleisli system, such as ACEDB [28], Entrez [24], Locus Link, UniGene, dbSNP, OMIM, PDB, SCOP [20], TIGR, KEGG, Medline, etc. For each of these sources, Kleisli typically provides many access functions corresponding to different capabilities of the sources. For example, Kleisli provides about 70 different but systematically organized functions to access and extract information from Entrez.
- Patent databases: Currently only access to the USPTO is supported.
- Interfaces: The Kleisli system also provides means for parsing input and writing output in HTML and XML formats. In addition, programming libraries are provided for Java and Perl to directly interface to Kleisli in a fashion similar to JDBC and ODBC. A graphical user interface called Discovery Builder is also available.

which retrieves GenPept reports matching a search string.

It is generally easy to develop a wrapper for a new data source, or modifying an existing one, and insert it into Kleisli. The main reason is that there is no impedance mismatch between the data model supported by Kleisli and the data model that is necessary to capture the data source. The wrapper is therefore often a very light-weight parser that simply parses records in the data source and prints it out in Kleisli's very simple data exchange format.

Example 7.1 *Let us consider the* `webomim-get-detail` *function used in Example 5.4. It uses an OMIM identifier to access the OMIM database and returns a set of objects matching the identifier. The output is of type*

```
{(#uid: num, #title: string, #gene_map_locus: {string},
  #alternative_titles: {string}, #allelic_variants: {string})}
```

Note that is this a nested relation: it is a set of records, and each record has three fields that are also of set types, viz. `#gene_map_locus`, `alternative_titles`, *and* `allelic_variants`. *This type of output would definitely present a problem if we had to give it to a system based on the flat relational model, as we would need to arrange for the information in these three fields to be sent into separate tables.*

Fortunately, such a nested structure can be mapped directly into Kleisli's exchange format. So the wrapper implementor would only need to parse each matching OMIM records and to write it out in a format like this:

```
{(#uid: 189965,
  #title: "CCAAT/ENHANCER-BINDING PROTEIN, BETA; CEBPB",
  #gene_map_locus: "20q13.1",
  #alternative_titles: {"C/EBP-BETA",
                        "INTERLEUKIN 6-DEPENDENT DNA-BINDING PROTEIN; IL6DBP",
                        "LIVER ACTIVATOR PROTEIN; LAP",
                        "LIVER-ENRICHED TRANSCRIPTIONAL ACTIVATOR PROTEIN",
                        "TRANSCRIPTION FACTOR 5; TCF5"},
  #allelic_variants: {})}
```

Here, instead of needing to create separate tables to keep the sets nested inside each record, the wrapper would simply print the appropriate set brackets { and } to enclose these sets. Kleisli would automatically deal with them as they were handed over by the wrapper. This kind of parsing and printing is extremely easy to implement. Given in Figure 3 is the relevant chunk of Perl codes in the OMIM wrapper implementing `webomim-get-detail`. □

8 Optimizations

The Kleisli system has a fairly advanced query optimizer. The optimizations provided by this optimizer include (1) "monadic" optimizations which are derived from the equationaly theory of monads, such as vertical loop fusion; (2) context-sensitive optimizations which are those equations that are true only in special contexts and are generally relying on certain long-range relationships between subexpressions, such as the absorption of subexpressions in the `then`-branch of an `if-then-else` construct that are equivalent to the condition of the construct; (3) relational optimizations which are optimizations relating to relational database sources, such as the migration of projections, selections, and joins to the external relational database management system; and many other optimizations such as parallelism, code motion, selective introduction of laziness, etc. We discuss some of these optimizations now.

```
#!/usr/bin/perl
#
#....stuff for connecting to OMIM omitted...
#....<CMD> is the input stream to be parsed...
#
# default values
$section = "none"; $state = 0; $id = "";
# the main program
print "{\n";
while (<CMD>) {
  chomp;
  if (/dispomim.cgi.cmd=entry.*id=([0-9]+)/) {
    $state = 1; $id = $1; $section = "title"; $line = ""; }
  # look for keywords to being parsing sections
  elsif (($state==1) && (/a href=\"\" name=\"$id\_(.*?)\"/)) {
    $section = "$1"; $line = $_; }
  # parse title
  elsif (($section eq "title") && (/<SPAN CLASS="H3"><font/)) {
    $title = $_; $title =~ s/<.*?>//g; }
  # parse alternative titles
  elsif (($section eq "MIM") && (m-</p></em>(.*)</h4>-)) {
    $tmp = $1; @alts = split /<br>/, $tmp; $alternativeTitles = "";
    foreach $x (@alts) { $alternativeTitles .= "\"$x\", "; }
    $alternativeTitles =~ s/, $//; }
  # parse gene map location
  elsif (($section eq "TEXT") && ($line =~ /^Gene map locus *(.*)/)){
    $geneMapLocus = $1; $geneMapLocus =~ s/<.*?>//g; $line = ""; }
  # parsing for Allelic variants
  # each allelic varient will have it's own section
  # need to group the allelic variants accross sections
  elsif ($section eq "ALLELIC_VARIANTS") { $variantTitle = ""; }
  elsif ($section =~ /AllelicVariant/) {
    $_ = $line; s/<.*?>//g; s/.\d+ *//; $variantTitle .= "\"$_\", "; }
  elsif (($state==1) && ($section eq "CREATION_DATE")) {
    $state = 0; $variantTitle =~ s/, $//; $variantTitle =~ s/\",/\"\n/g;
    $alternativeTitles =~ s/\",/\"\n/g;
    print "(#uid: $id, #title: \"$title\", \n";
    print "#gene_map_locus: \"$geneMapLocus\", \n";
    print "#alternative_titles: { $alternativeTitles }, \n";
    print "#allelic_variants: { $variantTitle }) \n"; } }
print "};\n";
```

Figure 3: The Perl codes of the wrapper implementing the `webomim-get-detail` function of Kleisli. It demonstrates the ease of developing wrappers for handling data sources that contain nested objects.

8.1 "Monadic" optimizations

Let us now consider the restricted form of structural recursion which corresponds to the presentation of monads by Kleisli [27, 8]. It is the combinator $\bigcup\{f(x) \mid x \in R\}$ obeying the following three equations:

$$\begin{array}{rcl} \bigcup\{f(x) \mid x \in \{\}\} & = & \{\} \\ \bigcup\{f(x) \mid x \in \{o\}\} & = & f(o) \\ \bigcup\{f(x) \mid x \in A \cup B) & = & (\bigcup\{f(x) \mid x \in A\}) \cup (\bigcup\{f(x) \mid x \in B\}) \end{array}$$

This combinator is at the heart of the $\mathcal{NRC}$, the abstract representation of queries in the implementation of sSQL. It earns its central position in the Kleisli system because it offers tremendous practical and theoretical convenience. The direct correspondence in sSQL is: `select y from R x, f(x) y`. This combinator is a key operator in the library of complex object routines in Kleisli. All sSQL queries can be and are first translated into $\mathcal{NRC}$ via Wadler's identities [27, 8]. The practical convenience of the $ext(\cdot)(\cdot)$ combinator is best seen in query optimizations.

A well-known optimization rule is vertical loop fusion [16], which corresponds to the physical notion of getting rid of intermediate data and the logical notion of quantifier elimination. Such an optimization on queries in the comprehension syntax can be expressed informally as $\{e \mid G_1, ..., G_n, x \in \{e' \mid H_1, ..., H_m\}, J_1, ..., J_k\} \rightsquigarrow \{e[e'/x] \mid G_1, ..., G_n, H_1, ..., H_m, J_1[e'/x], ..., J_k[e'/x]\}$ Such a rule in comprehension form is very simple to grasp. Basically the intermediate set built by the comprehension $\{e' \mid H_1, ..., H_m\}$ has been eliminated, in favour of generating the x on the fly. In practice it is quite messy to implement the rule above. In writing that rule, the informal "..." denotes any number of generator-filters in a comprehension. When it comes to actually implementing it, a nasty traversal routine must be written to skip over the non-applicable G_i in order to locate the applicable $x \in \{e' \mid H_1, ..., H_m\}$ and J_i. Let us now consider the $\bigcup\{f(x) \mid x \in R\}$ combinator. Its effect on the optimization rule for vertical loop fusion is dramatic. This optimization is now expressed as $\{f(x) \mid x \in \bigcup\{g(y) \mid y \in R\}\} \rightsquigarrow \bigcup\{\bigcup\{f(x) \mid x \in g(y)\} \mid y \in R\}$ The informal and troublesome "..." no longer appears. Such a rule can be coded up straightforwardly in almost any implementation language.

In order to illustrate this point more concretely, it is necessary to introduce some detail from the implementation of the Kleisli system. Recall from the introductory section that Kleisli is implemented on top of the Standard ML of New Jersey (SML). The type `SYN` of SML objects that represent queries in Kleisli is declared as:

```
type VAR = int                        (* Variables, represented by int *)
type SVR = int                        (* Server connections, represented by int *)
type CO = ...                         (* Representation of complex objects *)
datatype SYN = ...
| EmptySet                            (* { }                    *)
| SngSet of SYN                       (* { E }                  *)
| UnionSet of SYN * SYN               (* E1 U E2                *)
| ExtSet of SYN * VAR * SYN           (* U{ E1 | \x <- E2 }  *)
| IfThenElse of SYN * SYN * SYN       (* if E1 then E2 else E3 *)
| Read of SVR * real * SYN            (* process E using S,
       the real is the request priority assigned by optimizer *)
| Variable VAR                        (* x *)
| Binary (CO * CO -> CO) * SYN * SYN (* Construct for caching
       static objects. This allows the optimizer to insert
       some codes for doing dynamic optimization *)
```

All SML objects that represent optimization rules in Kleisli are functions and they have type `RULE`:

```
type RULE = SYN -> SYN option
```

If an optimization rule r can be successfully applied to rewrite an expression e to an expression e', then $r(e) =$ `SOME`(e'). If it cannot be successfully applied, then $r(e) =$ `NONE`.

We return to the rule on vertical loop fusion. As promised earlier, we have a very simple implementation:

Example 8.1 *Vertical loop fusion.*

```
fun Vertfusion(ExtSet(E1,x,ExtSet(E2,y,E3))) = SOME(ExtSet(ExtSet(E1,x E2),y,E3))
| Vertfusion _ = NONE
```

□

8.2 Context-sensitive optimizations

The Kleisli optimizer has an extensible number of phases. Each phase is associated with a rule-base and a rule application strategy. A large number of rule application strategies are supported, such as `BottomUpOnce`, which applies rules to rewrite an expression tree from leaves to root in a single pass. By exploiting higher-order functions, all of these rule application strategies can be decomposed into a "traversal" component that is common to all strategies and a very simple "control" component that is special for each strategy. In short, higher-order functions can generate all these strategies extremely simply, resulting in a very small optimizer core. To give some ideas on how this is done, some SML code fragments from the optimizer module mentioned are presented below.

The "traversal" component is a higher-order function that is shared by all strategies:

```
val Decompose: (SYN -> SYN) -> SYN -> SYN
```

Recall that `SYN` is the type of SML object that represents query expressions. The `Decompose` function accepts a rewrite rule r and a query expression Q. Then it applies r to all immediate subtrees of Q to rewrite these immediate subtrees. Note that it does not touch the root of Q and it does not traverse Q—it just nonrecursively rewrites immediate subtrees using r. It is therefore very straightforward and looks like this:

```
fun Decompose r (SngSet N) = SngSet(r N)
| Decompose r (UnionSet(N,M)) = UnionSet(r N, r M)
| Decompose r (ExtSet(N,x,M)) = ExtSet(r N, x, r M)
| ...
```

A rule application strategy S is a function having the following type

```
val S: RULEDB -> SYN -> SYN
```

The precise definition of the type `RULEDB` is not important to our discussion at this point and is deferred until later. Such a function takes in a rule base R and a query expression Q and optimizes it to a new query expression Q' by applying rules in R according to the strategy S.

Assume that `Pick: RULEDB -> RULE` is a SML function that takes a rule base R and a query expression Q and returns `NONE` if no rule is applicable, and `SOME`(Q') if some rule in R can be applied to rewrite Q to Q'. Then the "control" components of all the strategies mentioned earlier can be generated in SML in a very simple way.

Example 8.2 *The* `BottomUpOnce` *strategy applies rules in a leaves-to-root pass. It tries to rewrite each node at most once as it moves towards the root of the query expression. Here is its "control" component:*

```
fun BottomUpOnce RDB Qry =
  let fun Pass SubQry =
        let val BetterSubQry = Decompose Pass SubQry
         in case Pick RDB BetterSubQry
           of SOME EvenBetterSubQry => EvenBetterSubQry
           | NONE => BetterSubQry end
  in Pass Qry end
```

□

Let us now present an interesting class of rules that requires the use of multiple rule application strategies. The scope of rules like the vertical loop fusion in the previous section is over the entire query. In contrast, this class of rules has two parts. The inner part is "context sensitive" and its scope is limited to certain components of the query. The outer part scopes over the entire query to identify contexts where the inner part can be applied. The two parts of the rule can be applied using completely different strategies.

A rule base RDB is represented in our system as an SML record of type

```
type RULEDB = {
   DoTrace: bool ref,
   Trace: (rulename -> SYN -> SYN -> unit) ref,
   Rules: (rulename * RULE) list ref }
```

The `Rules` field of RDB stores the list of rules in RDB together with their names. The `Trace` field of RDB stores a function f that is to be used for tracing the usage of the rules in RDB. The `DoTrace` field of RDB stores a flag to indicate whether tracing is to be done. If tracing is indicated, then whenever a rule of name N in RDB is applied successfully to transform a query Q to Q', the trace function is invoked as $f\ N\ Q\ Q'$ to record a trace. Normally, this simply means a message like "Q is rewritten to Q' using the rule N" is printed. However, the trace function f is allowed to carry out considerably more complicated activities.

It is possible to exploit trace functions to achieve sophisticated transformations in a simple way. An example is the rule that rewrites `if` e_1 `then` ... e_1 ... `else` e_3 to `if` e_1 `then` ... `true` ... `else` e_3. The inner part of this rule rewrites e_1 to `true`. The outer part of this rule identifies the context and scope of the inner part of this rule: limited to the `then`-branch. This example is very intuitive to a human being. In the `then`-branch of a conditional, all subexpressions that are identical to the test predicate of the conditional must eventually evaluate to `true`. However, such a rule is not so straightforward to express to a machine. The informal "..." are again in the way. Fortunately, rules of this kind are straightforward to implement in our system.

Example 8.3 *The If-then-else absorption rule is expressed by the* `AborbThen` *rule below. The rule has three clauses. The first clause says that the rule should not be applied to an* `IfThenElse` *whose test predicate is already a Boolean constant, because it would lead to non-termination otherwise. The second clause says that the rule should be applied to all other forms of* `IfThenElse`*. The third clause says that the rule is not applicable in any other situation.*

```
fun AbsorbThen (IfThenElse(Bool _,_,_)) = NONE
| AbsorbThen (IfThenElse(E1,E2,E3)) =
    let fun Then E = if SyntaxTools.Equiv E1 E then SOME(Bool true) else NONE
    in case ContextSensitive Then TopDownOnce E2
       of SOME E2' => IfThenElse(E1,E2',E3)
       | NONE => NONE end
| AbsorbThen _ = NONE
```

The second clause is the meat of the implementation. The inner part of the rewrite `if` e_1 `then` ... e_1 ... `else` e_3 *to* `if` e_1 `then` ... `true` ... `else` e_3 *is captured by the function* `Then` *which rewrites any* e *identical to* e_1 *to* `true`*. This function is then supplied as the rule to be applied using the TopDownOnce strategy within the scope of the* `then`*-branch ...* e_1 *... using the* `ContextSensitive` *rule generator given below.*

```
fun ContextSensitive Rule Strategy Qry =
let val Changed = ref false                                    (* This flag is set if Rule is applied *)
   val RDB = {                                                 (* Set up a context-sensitive rule base *)
     DoTrace = ref true,
     Trace = ref (fn _ => fn _ => fn _ => Changed := true) (* Changed is true if Rule is used *)
     Rules = ref [("", Rule)]}
   val OptimizedQry = Strategy RDB Qry                         (* Apply Rule using Strategy. *)
in if !Changed then SOME OptimizedQry else NONE end
```

This `ContextSensitive` *rule generator is reused for many other context-sensitive optimization rules, such as the rule for migrating projections to external relational database systems to be presented shortly.* □

8.3 Relational optimizations

Relational database systems are the most powerful data sources that Kleisli interfaces to. These database systems are themselves equipped with the ability to perform sophisticated transformations expressed in SQL. A good optimizer should aim to migrate as many operations in Kleisli to these systems as possible. There are four main optimizations that are useful in this context: the migration of projections, selections, and joins on a single database; and the migration of joins across two databases. The Kleisli optimizer has four different rules to exploit these four opportunities. We show one of them below.

Let us consider the rule for migrating projections. A special case of this rule is to rewrite `select x.name from (process "select * from T" using A) x` to `select x.name from (process "select name from T" using A) x`, where we use `process Q using A` to denote sending a SQL query `Q` to a relational database `A`. In the original query, the entire table `T` has to be retrieved. In the rewritten query, only one column of that table has to be retrieved. More generally, if `x` is from a relational database system and every use of `x` is in the context of a field projection `x.l`, these projections can be "pushed" to the relational database so that unused fields are not retrieved and transferred.

Example 8.4 *The rule for migrating projections to a relational database is implemented by* `MigrateProj` *below. The rule requires a function* `FullyProjected` x N *that traverses an expression* N *to determine whether* x *is always used within* N *in the context of a field projection and to determine what fields are being projected; it returns* `NONE` *if* x *is not always used in such a context; otherwise, it returns* `SOME` L*, where the list* L *contains all the fields being projected. This function is implemented in a simple way using the* `ContextSensitive` *rule generator from Example 8.3.*

```
fun FullyProjected x N =
    let val (Count, Projs) = (ref 0, ref [])
        fun FindProjs (Variable y) = (if x = y then inc Count else (); NONE)
        | FindProjs (Proj (L, Variable y)) =
           (if x = y then Projs := L :: (!Projs) else (); NONE)
        | FindProjs _ = NONE
    in ContextSensitive FindProjs BottomUpOnce N;
       if length (!Projs) = !Count then SOME (!Projs) else NONE
    end
```

The `MigrateProj` *rule is defined below. The function* `SQL.PushProj` *is one of the many support routines available in the current release of Kleisli that handle manipulation of SQL queries and other* `SYN` *abstract syntax objects.*

```
fun MigrateProj (ExtSet (N, x, Read (S, p, String M))) =
   if Annotations.IsSQL S        (* test if S connects to a SQL server *)
   then case FullyProjected x N  (* test if x is always in a projection *)
       of SOME Projs => SOME (ExtSet (N, x, Read (S, p, String (SQL.PushProj Projs M))))
       | NONE => NONE
   else NONE
| MigrateProj _ = NONE
```

□

Besides the four migration rules above, there is also a rule for reordering joins on two relational databases. In addition, there are also rules for parallelizing queries, for large-scale code motion, and so on. We omit a description due to space constraint.

9 User Interfaces

Kleisli is also equipped with application programming interfaces for use with Java and Perl. It also has a graphical interface for non-programmers. These interfaces are described in this section.

9.1 Programming language interface

The high-level query language, sSQL, of the Kleisli system was designed to express traditional (nested relational) database-style queries. Not every query in bioinformatics falls into this class. For these non-database-style queries, some other programming languages can some time be a more convenient or more efficient means of implementation. Therefore, we develop the Pizzkell suite [31] of interfaces to the Kleisli Exchange Format for various popular programming languages. Each of these interfaces in the Pizzkell suite is a library package for parsing data in Kleisli's exchange format into an internal object of the corresponding programming language. It also serves as a means for embedding the Kleisli system into that programming language, so that the full power of Kleisli is available within that programming language. The Pizzkell suite currently include CPL2Perl and CPL2Java, for Perl and Java. We describe CPL2Perl.

In contrast to sSQL in Kleisli, which is a high-level interface that comes with a sophisticated optimizer and other database-style features, CPL2Perl has a different purpose and is at a lower level. Whereas sSQL is aimed at extraction, integration, and preparation of data for analysis, CPL2Perl is intended to be used for implementing analysis and textual formatting of the prepared data in Perl. Thus, we designed CPL2Perl as a Perl module for parsing data conforming to the data exchange format of Kleisli into native Perl objects.

The main functions in CPL2Perl are divided into three packages:

1. The RECORD package simulates the record data type of the Kleisli Exchange Format by using a reference of Perl's hash. Some functions are defined in this package:
   - `new` is the constructor of a record. For example, to create a record such as `(#anno_name: "db_xref", #descr: "taxon:10090")` in a Perl program, we write:

```
$rec = RECORD->new ("anno_name",
  "db_xref", "descr","taxon:10090");
```

where `$rec` becomes the reference of this record in the Perl program.

- `Project` gets the value of a specified field in a record. For example,

  ```
  $rec->Project("descr");
  ```

  will return the value of field `#descr` in the record referenced by `$rec` in the Perl program.

2. The LIST package simulates the list, set, and bag data type in the Kleisli Data Exchange Format. All these three "bulk" data types are be converted as a reference of Perl's list. Its main function is:

   - `new` is the constructor of a "bulk" data such as a list, a bag, or a set. It works the same way as you would initialize a list in Perl. Eg.,

     ```
     $l = LIST->new ( "tom", "jerry" );
     ```

     where `$l` will be the reference of this list in the Perl program.

3. The CPLIO package provides the interface to read data from a Kleisli-formatted data file or pipe directly. It supports both eager and lazy access methods. Some functions in this package are:

   - `Open1`. This function opens the specified Kleisli-formatted data file and returns the handle of this file in Perl. Note that its supports all the input-related features of the usual "open" operation of Perl, including the use of pipes. For example, to open Kleisli-formatted file `"sequences.val"`, we write:

     ```
     $hd = CPLIO->Open1("sequences.val");
     ```

   - `Open1a` is another version of `Open1` function which can take a string as input stream. The first parameter specified the child process you want to execute and the second parameter is input string. Here is an example that calls Kleisli from CPL2Perl to extracts accession numbers from a sequence file:

     ```
     $cmd = qq{
     create view X from sequences using stdin;
     select x.accession from X x; };
     $a= CPLIO->Open1a ( "./ssql" , $cmd);
     ```

   - `Open2` differs from `Open1a` in that it allows a program to communicate in both directions with Kleisli or other systems. Its parameter is the Kleisli or other systems to call. It returns a list. The first element is a reference of CPLIO object and the second one is an Input stream which we can send our request into. eg:

     ```
     ($a, $b) = CLPIO->Open2("./ssql");
     print $b $cmd1; flush $b; $a->Parse;
     ...
     print $b $cmd2; flush $b; $a->Parse;
     ...
     ```

   - `Parse` is a function that reads all the data from an opened file until it can assemble a complete object or a semicolom(;) is found. The return value will be the reference of the parsed object in Perl. For example, assume that our opened file is a set of records having a `#accession` field, we can print the values of this field like this:

     ```
     $set = $hd->Parse;
     foreach $rec (@{$set} ) {
     $n = $rec->Project("accession");
     print "$n\n"; }
     ```

   - `LazyRead` is a function to read data from an opened file lazily. We normally use this function if the data type in the opened file is a set, bag, or list. This function only reads one element into memory each time. Thus, if our opened file is a very big set, `LazyRead` is just the right function to access the records and print accession numbers:

```
while (1) {
$rec = $hd->LazyRead; last if ($rec eq "");
$n = $rec->Project("accession");
print "$n\n"; }
```

We demonstrate the use of CPL2Perl for interfacing Kleisli to the Graphviz system, which is needed for the Protein Interaction Extraction System described in [32]. Graphviz[15] is a system for automatic layout of directed graphs. It accepts a general directed graph specification which is in essence a list of arcs of the form $x \to y$, which specifies an arc is to be drawn from the node x to the node y.

Example 9.1 *Assume that Kleisli produces a file* `$SPEC` *of type* `{(#actor: string, #interaction: string, #patient: string)}` *which describes a protein interaction pathway. These are records that say which "actor" inhibit or activate which "patient". We wish to build a Perl module* `MkGif` *that accepts this file and converts it into a directed graph specification and invokes Graphviz to layout and draws it as a GIF file* `$GIF`*. Here is the relevant parts of the Perl implementation of* `MkGif` *using CPL2Perl:*

```
use cpl2perl;
$a = CPLIO->Open1 ("$SPEC");
open (DOT, "| ./dot -Tgif  > $GIF");
print DOT "digraph aGraph {\n";
while (1) {
 $rec = $a->LazyRead; last if ($rec eq "");
 $start = $rec->Project ("actor");
 $end = $rec->Project ("patient");
 $type = $rec->Project ("interaction");
 if ($type eq "inhibit") {
   $edgecolor = "red"; }
 else {
   $edgecolor = "green"; }
 $edge = "[color = $edgecolor]";
 print DOT " $start-> $end $edge;\n"; }
print DOT "};\n";
$a->Close;
close (DOT);
```

The first 3 lines establishes connections to the Kleisli file `$SPEC` *and to the Graphviz program* `dot`*. The next few lines use the* `LazyRead` *and* `Project` *functions of CPL2Perl to extract each interaction record from the file and to format it for Graphviz to process. Upon finishing the layout computation, Graphviz draws the interaction pathway into the file* `$GIF`*.* □

This example, though short, demonstrates how CPL2Perl smoothly integrates the Kleisli Exchange Format into Perl. This greatly facilitates both the development of data drivers for Kleisli and the development of down-stream processing (such as pretty printing) of results produced by Kleisli.

9.2 Graphical interface

There is also a graphical interface to the Kleisli system that is designed for non-programmers. It is called the Discovery Builder and is developed by folks at geneticXchange Inc. This graphical interface makes it far easier to visualize the source data required to formulate the queries and generates the necessary sSQL codes. It allows a user to see all available data sources and their associated metadata and assists the user to navigate and to

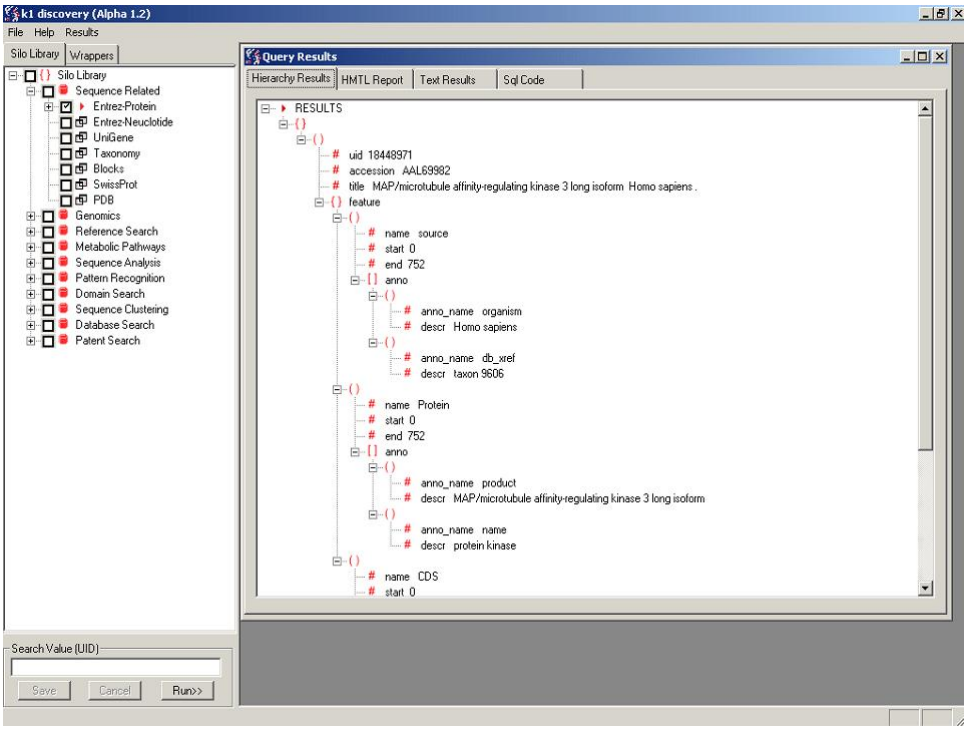

Figure 4: A screenshot of the Discovery Builder, a graphical interface to Kleisli.

specify his query on these sources. A screenshot of Discovery Builder is presented in Figure 4. It provides all of the following key functions:

- A graphical interface that can "see" all of the relevant biological data sources, including metadata—tables, columns, descriptions, ...—and then construct a query "as if" the data were local.
- Add new wrappers for any public or proprietary data sources, typically within hours, and then have them enjoined in any series of ad-hoc queries that can be created.
- Execute the queries, which may join many data sources that can be scattered all over the globe, and get fresh result data quickly.

10 Comparison

Let us compare the Kleisli system to other approaches to bioinformatics data integration problems. The alternatives include SRS [14], Discovery Link [17], OPM [10], etc. SRS is marketed by LION and is basically an IR system. Indices are built on a local copy of the source data. One accesses the data by supplying suitable keywords. All records matching those keywords are returned. The system more or less stops there. If one wishes to perform further operations or transformations on the results, then one would need to perform them outside of SRS by writing additional programs.

Discovery Link is marketed by IBM and, in principle, it goes one step beyond SRS. It provides a high-level query language that allows one to perform further manipulations on the initial results. However, in practice, it still has some way to go. The reason is that Discovery Link is tied to the flat relational model. That is, every piece of data that it handles must be a table of atomic objects like strings and numbers. Unfortunately, as discussed earlier, most of the data sources in biology are deeply nested. Therefore, there is a severe impedance mismatch between these sources and Discovery Link. For example, to put the Swissprot database [3] into a relational database in third normal form would require us to break every Swissprot record into over 20 pieces! This normalization process requires a lot of skill. Similarly to query the transformed source in Discovery Link requires some mental and performance overhead. In short, it is difficult to extend Discovery Link with new sources. In addition, Discovery Link cannot store nested objects in a natural way.

OPM goes one step beyond Discovery Link. It supports the equivalent of nested relational model and thus removes the impedance mismatch. It however requires the use of a global integrated schema and it also stores the nested relations by automatically converting them into third normal form—which leads to a conceptual record being broken up into many pieces when stored. Having a global schema is an added design overhead, especially when some of the underlying sources have no schema or extremely large schema, as it tends to be brittle if the underlying sources evolve. Therefore, it may be costly to extend OPM with new sources. The conversion to third normal form to store nested objects, even though transparent to the user, can also result in performance problems. OPM was marketed by Gene Logic, but it sales was discontinued some time ago.

Kleisli goes one step beyond OPM. It supports nested relations. It has a high-level query language to manipulate these relations. It does not need any schema. It supports a simple exchange format so that new sources can be inserted with ease. It has a good query optimizer. It does not store nested relations by fragmenting them. Kleisli does not have its own native database management system. Instead, Kleisli has the ability to turn many kinds of database systems into an update-able store conforming to its complex object data model. We therefore believe that Kleisli is very close to the ideal data integration solution.

11 Conclusions

In the era of genome-enabled large-scale biology, high-throughput technologies from DNA sequencing, microarray gene expression, mass spectroscopy, to combinatory chemistry and high-throughput screening have generated an unprecedented volume and diversity of data. These data are deposited in disparate specialized geographically dispersed databases that are heterogeneous in data formats and semantic representations. In parallel, there is a rapid proliferation of computational tools and scientific algorithms for data analysis and knowledge extraction. The challenge to life science today is how to process and integrate this massive amount of data and information for research and discovery. The heterogeneous and dynamic nature of biomedical data sources presents a continuing challenge to accessing, retrieving, and integrating information across multiple sources.

Many features of the Kleisli system [12, 29, 30] are particularly suitable for automating the data integration process. Kleisli employs a distributed and federated approach to access external data sources via the wrapper layer, and thus can access the most up-to-date data on demand. Kleisli provides a complex nested internal data model that encompasses most of the current popular data models including flat files, HTML, XML, relational databases, and thus serves as natural data exchanger for different data formats. Kleisli offers a robust query optimizer and a powerful and expressive query language to manipulate and transform data, and thus facilitates data integration. Finally, Kleisli has the capability of converting relational database management systems such as Sybase, MySQL, Oracle, DB2, Informix, etc. into nested relational stores, and thus enabling the creation of robust warehouses of complex biomedical data. Leveraging the capabilities of Kleisli leads to the development of the query scripts that give us a high-level abstraction beyond low-level codes to access a combination of the relevant data and the right tools to solve the right problem.

The Kleisli system and its high-level query language sSQL embody many advances that have been made in database query languages and in functional programming. It has made significant impact on data integration in bioinformatics. Indeed, since the early Kleisli prototype was applied to bioinformatics, it has been used to efficiently solve many bioinformatics data integration problems. To date, thanks to the use of its high-level query language, we do not know of another system that can express general bioinformatics queries as succinctly as Kleisli.

Let us close this chapter by summarizing the key ideas behind the success of the Kleisli system. The first is its use of a complex object data model where sets, bags, lists, records and variants can be flexibly combined. The second is its use of a high-level query language that allows these objects to be easily manipulated. The third is its use of a self-describing data exchange format, which serves as a simple conduit to external data sources. The fourth is its query optimizer, which is capable of many powerful optimizations.

12 Acknowledgments

We would like to acknowledge the contributions to Kleisli by our colleagues. The first prototype of Kleisli was designed and implemented in 1994, while Wong was at the University of Pennsylvania. Peter Buneman, Val Tannen, Leonid Libkin, and Dan Suciu contributed to the query language theory and foundational issues of Kleisli. Chris Overton introduced us to problems in bioinformatics. Kyle Hart helped us in applying Kleisli to address the first bioinformatic integration problem ever solved by Kleisli. Wong re-designed and re-implemented the entire system in 1995, when he returned to the Institute of Systems Science (now renamed Kent Ridge Digital Labs, following its corporatization.) The new system, which is in production use in the pharmaceutical industry, has many new implementation ideas, has much higher performance, is much more robust, and has much better support for bioinformatics. Desai Narasimhalu supported its development in Singapore. Oliver Wu and Jiren Wang added much to its bioinformatics support under funding from the Singapore Economic Development Board. Finally, the folks at geneticXchange Inc were responsible for greatly improving Kleisli and for taking it to the market; please visit them at www.geneticxchange.com.

References

[1] S. F. Altschul and W. Gish. Local alignment statistics. *Methods in Enzymology*, 266:460–480, 1996.

[2] S. F. Altschul et al. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[3] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its supplement TrEMBL in 1999. *Nucleic Acids Research*, 27(1):49–54, 1999.

[4] P. G. Baker et al. TAMBIS—transparent access to multiple bioinformatics information sources. *Intelligent Systems for Molecular Biology*, 6:25–34, 1998.

[5] V. Tannen et al. Structural recursion as a query language. In *Proc. 3rd International Workshop on Database Programming Languages*, pages 9–19, 1991.

[6] V. Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proc. 18th International Colloquium on Automata, Languages, and Programming*, pages 60–75, 1991.

[7] P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[8] P. Buneman et al. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[9] C. Burks et al. GenBank. *Nucleic Acids Research*, 20 Supplement:2065–9, 1992.

[10] I.M.A. Chen et al. An overview of the Object-Protocol Model (OPM) and OPM data management tools. *Information Systems*, 20)5):393–418, 1995.

[11] E. F. Codd. A relational model for large shared data bank. *Communications of the ACM*, 13(6):377–387, 1970.

[12] S. Davidson et al. BioKleisli: A digital library for biomedical researchers. *International Journal of Digital Libraries*, 1(1):36–53, 1997.

[13] G. Dong et al. Local properties of query languages. In *Proc. 6th International Conference on Database Theory*, pages 140–154, 1997.

[14] T. Etzold and P. Argos. SRS: Information retrieval system for molecular biology data banks. *Methods Enzymol.*, 266:114–128, 1996.

[15] E.R. Gansner et al. A technique for drawing directed graphs. *IEEE Trans. Software Engineering*, 19(3):214–230, 1993.

[16] A. Goldberg and R. Paige. Stream processing. In *Proc. ACM Symposium on LISP and Functional Programming*, pages 53–62, 1984.

[17] L.M. Haas et al. DiscoveryLink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.

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

[19] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.

[20] A. Murzin et al. SCOP: A structural classification of protein database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540, 1995.

[21] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD. *NCBI ASN.1 Specification*, 1992. Revision 2.0.

[22] P. Pearson et al. The GDB human genome data base anno 1992. *Nucleic Acids Research*, 20:2201–2206, 1992.

[23] C. Schoenbach et al. FIMM, a database of functional molecular immunology. *Nucleic Acids Research*, 28(1):222–224, 2000.

[24] G. D. Schuler et al. Entrez: Molecular biology database and retrieval system. *Methods in Enzymology*, 266:141–162, 1996.

[25] D. Suciu. Bounded fixpoints for complex objects. *Theoretical Computer Science*, 176(1–2):283–328, 1997.

[26] D. Suciu and L. Wong. On two forms of structural recursion. In *LNCS 893: Proc. 5th International Conference on Database Theory*, pages 111–124, 1995.

[27] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[28] S. Walsh et al. ACEDB: A database for genome information. *Methods Biochem Anal*, 39:299–318, 1998.

[29] L. Wong. The functional guts of the kleisli query system. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, pages 1–10, 2000.

[30] L. Wong. Kleisli, a functional query system. *J. Funct. Prog.*, 10(1):19–56, 2000.

[31] L. Wong. Kleisli, its exchange format, supporting tools, and an application in protein interaction extraction. In *Proc. 1st IEEE International Symposium on Bio-Informatics and Biomedical Engineering*, pages 21–28, 2000.

[32] L. Wong. PIES, a protein interaction extraction system. In *Proc. Pacific Symposium on Biocomputing*, pages 520–531, 2001.