J. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom

Kleisli, a Functional Query System

LIMSOON WONG

Kent Ridge Digital Labs
21 Heng Mui Keng Terrace, Singapore 119613
Email: limsoon@krdl.org.sg

Abstract

Kleisli is a modern data integration system that has made a significant impact on bioinformatics data integration. This paper contains a brief introduction to the Kleisli system and an example to illustrate its uses in the bioinformatics arena. The primary query language provided by Kleisli is called CPL, which is a functional query language whose surface syntax is based on the comprehension syntax. Kleisli is itself implemented using the functional language SML. So this paper also describes the influence of functional programming research that benefits the Kleisli system, especially the less obvious ones at the implementation level.

Availability. Kleisli has been commercialized under the name "KRIS". It is available from *Kris Technology Inc., 713 Santa Cruz Ave, #2, Menlo Park, CA 94025. Direct email to* info@kris-inc.com *and web browser to* http://www.kris-inc.com.

1 Introduction

The Kleisli system (Davidson *et al.*, 1997) is an advanced broad-scale integration technology that has proved useful in the bioinformatics arena (Benton, 1996; Karp, 1996; Baker & Brass, 1998; Leser *et al.*, 1998). Many bioinformatics problems require access to data sources that are high in volume, highly heterogeneous and complex, constantly evolving, and geographically dispersed. Solutions to these problems usually involve multiple carefully sequenced steps and require information to be passed smoothly between the steps. Kleisli is designed to handle these requirements directly by providing a high-level query language, CPL (Wong, 1998a), that can be used to express complicated transformation across multiple data sources in a clear and simple way.

Many key ideas in the Kleisli system are influenced by functional programming research, as well as database query language research. Its high-level query language CPL is 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/NJ). Even the data format that Kleisli uses to exchange information with the external world is derived from an idea in type inference.

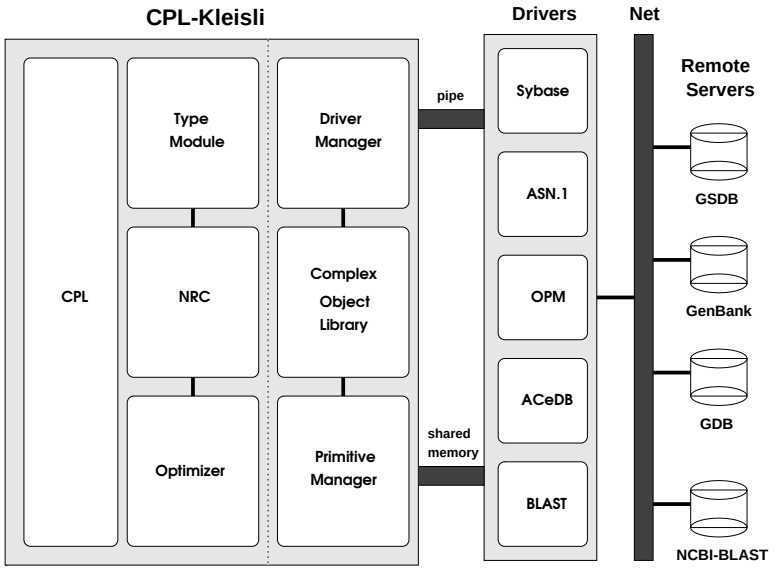
This paper provides an overview of the Kleisli system; a summary of its impact

on query language theory; an illustration of its uses in the bioinformatics arena; and a description of the influence of functional programming research that benefits the Kleisli system, especially the less obvious ones at the implementation level. The organization of the paper is as follows. Section 2 has three subsections that give an overview of the architecture, data model, and query language (CPL) of Kleisli. Section 3 has three subsections that explain the impact of Kleisli on the integration of biology data and illustrate this impact using a real-life example. Section 4 also has three subsections that single out three areas in the implementation of Kleisli and discuss how they are influenced by functional programming research. In particular, how type inference gives rise to Kleisli's self-describing data exchange format, how monad gives rise to Kleisli's internal abstract representation of queries and simple optimization rules, and how higher-order functions give rise to a simple implementation of Kleisli's powerful optimizer. The effectiveness of this optimizer is discussed in Section 5. Section 6 discusses the differences between the old 1994 prototype the author built at the University of Pennsylvania and the current production version constructed at Kent Ridge Digital Labs as presented in this paper.

2 Quick Tour of Kleisli

Kleisli can be regarded as a system for data access, transformation, and integration. It sits between heterogeneous data sources and tools that operate on some integrated version of the available data. In this quick tour, the focus is on the data model of Kleisli that is based on complex object types and on the high-level query language supported by Kleisli called CPL, which stands for Collection Programming Language. Its basic architecture is presented in Subsection 2.1, its data model in Subsection 2.2, and an overview of CPL in Subsection 2.3.

2.1 Architecture



The Kleisli system(Davidson *et al.*, 1997) is written entirely in SML/NJ. The architecture of the system is depicted in the figure above. Kleisli is extensible in many ways: It can be used to support many other high-level query languages by replacing the CPL module. Kleisli can also be used to support many different types of external data sources by adding new drivers, which forward Kleisli's requests to these sources and translate their replies into Kleisli's exchange format. The installation at Kent Ridge Digital Labs contains over sixty drivers for all popular bioinformatics systems, including Sybase, Oracle, Entrez (Schuler *et al.*, 1996), WU-BLAST2 (Altschul & Gish, 1996), Gapped BLAST (Altschul *et al.*, 1997), ACEDB (Walsh *et al.*, 1998), etc. Furthermore, the optimizer of Kleisli can be customized by adding different rules and strategies.

When a query is submitted to Kleisli, it is first processed by the CPL Module which translates it into an equivalent expression in NRC. The abstract calculus NRC is based on that described in (Buneman *et al.*, 1995), 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 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 drivers and external primitives as needed through pipes or shared memory. The Driver and Primitive Managers keep information on external sources and primitives and the wrapper/interface codes to them. The Complex Object Library contains routines for manipulating complex objects such as codes for set intersection and codes for iterating over a set.

The core of the Kleisli system is divided into two main components, as shown by the dotted line in the figure. The first component provides high-level language support and consists of the CPL Module (approximately 7000 lines of SML codes), the Type Module (approximately 3100 lines), the Nested Relational Calculus (NRC) Module (approximately 3300 lines), and the Optimizer Module (approximately 3500 lines including codes for actual optimization rules). The second component is the query engine and consists of the Driver Manager (approximately 16100 lines including codes for actual drivers), the Primitive Manager (approximately 5400 lines including codes for actual primitives implementation), and the Complex Object Library (approximately 7000 lines). More detail of the implementation of Kleisli is deferred to a later section, where the influence of functional programming is discussed. For now, let us proceed to a quick tour of CPL and see some bioinformatic actions.

2.2 Complex Object Types

The data model underlying Kleisli is a complex object type system that goes beyond the "sets of records" or "flat relations" type system of relational databases (Codd, 1970). It allows arbitrarily nested records, sets, lists, bags, and variants. A bag is also called a multi-set, which is conceptually a set in which duplicates may occur. A list is conceptually a bag with order. A variant is also called a tagged union type

and represents a type that is "either this or that". Our 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.

The simultaneous availability of sets, bags, and lists in Kleisli deserves some comments. In a relational database, the sole "bulk" data type is the set. In a functional programming language, the usual "bulk" data type is the list. Having only one bulk data type presents at least two problems in many important real life applications. Firstly, the particular bulk data type, be it set or list, may not be a natural model of real data. For example, if we are modeling the author list of a paper and the ordering of authors is important, it is conveniently modeled as a list. If it is modeled as a set, then it is necessary to model it as a set of author-position pairs, to avoid losing information on the ordering of authors. Secondly, the particular bulk data type, be it set or list, may not be an efficient model of real data. For example, if we are modeling the author list of a paper and the ordering of authors is unimportant for the particular application we have in mind, it is conveniently modeled as a set. If it is modeled as a list, then well-known database query optimizations such as re-ordering of joins (Ullman, 1989) can no longer be applied, as they normally do not preserve positional ordering.

Example 2.1
The GenPept report is the format chosen by the US National Centre for Biotechnology Information to present amino acid sequence information. The information includes the sequence itself, its interesting biological features, its derivation from a source DNA sequence, references to papers and authors studying it, cross references to other databases containing related information, and so on. While an amino acid sequence can be thought of as a string of letters, certain regions and positions of the string are of special biological interest, such as binding sites, active 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 (Schuler *et al.*, 1996).

```
(#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 it is a record of set of lists of records. Here is the detail. It is a record of 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 respectively 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: $t ::=$ `num` | `string` | `bool` |

{t} | {|t|} | [t] | ($l_1 : t_1$, ..., $l_n : t_n$) | <$l_1 : t_1$, ..., $l_n : t_n$>. 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 <$l_1 : t_1$, ..., $l_n : t_n$> 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 constructed in CPL 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 2.2
A value conforming to our feature table type is

```
(#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")]), ...})
```

This particular feature table belongs to GenPept report 131470 and is that of a protein tyrosine phosphatase 1C sequence. The particular feature displayed above is from amino acid 0 to amino acid 594, which is actually the entire sequence. The feature entry displayed above has two annotations. The first indicates that this amino acid sequence is derived from mouse DNA sequence. The second provides a cross reference to the US National Center for Biotechnology Information taxonomy database, where further information on the taxonomic relationship of mouse to other organisms can be found. □

The schemas and structures of all popular bioinformatics databases, flat files, and softwares are easily mapped into this data model. At the extreme of data structure complexity are Entrez (Schuler *et al.*, 1996) and ACEDB (Walsh *et al.*, 1998), which contain deeply nested mixtures of sets, bags, lists, records, and variants. At the other extreme of data structure complexity are the relational database systems (Codd, 1970) such as Sybase and Oracle, which contain flat sets of records. Currently, Kleisli gives access to over sixty of these and other sources used regularly by molecular biology laboratories in Singapore and in phamaceutical companies. 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. So 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.

It would not be possible to map these sources so easily onto a relational database system or a deductive database system, as all relational and deductive database systems impose the first normal form requirement. The first normal form is an important concept of relational databases and is the basis of their practical implementation. It is also a key ingredient in guaranteeing termination of queries in deductive databases. A value is in first normal form if it is "flat", that is, it contains

no nested records, nested sets, or other "bulk" types. Relational database systems and their deductive extensions are designed to only manipulate data in first normal form or its further restrictions and their implementations exploit this first normal form assumption to achieve great efficiency.

The number of type constructors may seem spartan compared to those of popular functional programming languages, where arbitrary number of fresh types and type constructors can be introduced by a programmer. The closest database correspondence to types in programming languages is the schemas. A schema of a relational database contains a list of relation names, the structure of the underlying relations, and other information such as the presence of indices. The relation structures correspond to the definition of the types. In spite of this correspondence, there seems to be a different attitude towards types between the two worlds. In a database query language, it is not necessary to introduce a new relation type explicitly. Almost every query in a database query language results in a "new" type. For example, a projection query that extracts two fields from an existing relation having three fields in principle introduces a "new" record type having only those two named fields. In a functional programming language such as Haskell, new record types cannot be created easily, as they must be explicitly introduced before hand. So to implement the same query, the Haskell programmer would have to first introduce a new type with a type constructor having the two named fields. Even in a functional programming language like SML/NJ, where new record types can be used without prior type declaration, it is often the case that variant types must be declared before hand. Therefore, the number of types in databases is not spartan.

The difference is that the richness of types in database programming is hidden by their being implicit, while the richness of types in functional programming is highlighted by their being explicit. The fact that almost all database queries introduce "new" types makes a compelling reason for database programming languages to favour more flexible type constructors and types that are completely, conveniently, and anonymously defined in terms of their structures. Interestingly, more experimental or theoretical investigations of functional programming languages such as (Remy, 1989), also explored these ideas and favoured similar spartan but more flexible type constructions.

2.3 Collection Programming Language

The syntax of CPL is similar to that of the ODMG standard for object-oriented database languages (Cattell, 1996). An interesting feature of the syntax of CPL is the heavy used of the comprehension syntax, which showed up long ago in functional programming languages such as Miranda and later formalized by Wadler (1992). A typical comprehension in CPL syntax is `{ x * x | \x <- S, odd(x) }` which returns a set consisting of the squares of all odd numbers in the set `S`. This is similar to the notation found in functional languages, the main difference being that the binding occurrence of `x` is indicated by preceding it with a backslash, and that the expression returns a set rather than a list. As in functional languages, `\x <- S` is called a "generator", and `odd(x)` is called a "filter." Rather than giving the

complete syntax, which can be found in (Wong, 1998a), we illustrate CPL through a series of examples.

Example 2.3
This query extracts the titles and features of a set `DB` of feature tables. Note the use of `\x` to introduce the variable `x`. The effect of `\x <- DB` is to bind `x` to each element of the set `DB`.

```
{ (#title: x.#title, #feature: x.#feature) | \x <- DB };
```
□

Example 2.4
This query extracts the titles and features of those elements of `DB` whose titles contain `tyrosine` as a substring.

```
{ (#title: x.#title, #feature: x.#feature)
| \x <- DB, x.#title string-islike "%tyrosine%" };
```
□

These queries are no more than simple project-select queries. 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 are not in first normal form, these queries could be expressed in a relational query language. However, CPL can perform more complex restructurings such as nesting and unnesting not found in common relational database languages like SQL, as shown in the following examples.

Example 2.5
This query reduces the nested relation `DB`, which is a set of sets of lists, by one level, to a set of lists. It also restructures it by eliminating uids and accessions.

```
{ (#title: x.#title, #feature: f.#name, #start: f.#start, #end: f.#end,
   #annotations: f.#anno)
| \x <- DB,
  \f <- x.#feature };
```
□

Example 2.6
This query flattens `DB` completely. The `\a <--- f.#anno` has similar meaning to `\x <- DB`, but works on list instead of set. Thus it binds `a` to each item in the list `f.#anno`.

```
{ (#title: x.#title, #feature: f.#name, #start: f.#start, #end: f.#end,
   #anno-name: a.#anno_name, #anno-descr: a.#descr)
| \x <- DB,
  \f <- x.#feature,
  \a <--- f.#anno };
```
□

Example 2.7
This query navigates through each annotation `a` of each feature `f` of each entry `x` in `DB` to discover the organism of that entry. It then restructures the nested relation into a database of entries with associated organisms.

```
{ (#entry: x, #organism: a.#descr)
| \x <- DB,
  \f <- x.#feature,
  \a <--- f.#anno, a.#anno_name = "organism" };
```

□

Example 2.8
This query demonstrates how to do nesting in CPL. The subquery `DB'` is the restructuring of `DB` as in Example 2.7 above. The subquery `ORG` then extracts all organism names. The main query groups entries in `DB'` by organism names. It also sorts the output list by alphabetical order of organism names, because `[ u | \u <- ORG]` converts the set `ORG` into a duplicate-free sorted list.

```
let \DB' == { (#entry: x, #organism: a.#descr)
            | \x <- DB, \f <- x.#feature,
              \a <--- f.#anno, a.#anno_name = "organism" } in
let \ORG == { y.#organism | \y <- DB' }
in [ (#organism: z,
      #entries: { v.#entry | \v <- DB', v.#organism = z })
   | \z <--- [ u | \u <- ORG ] ];
```

□

The inspiration for CPL came primarily from (Breazu-Tannen *et al.*, 1991) that presented structural recursion as a query language. However, structural recursion has two difficulties. The first is that not every syntactically acceptable structural recursion program is logically well-defined (Breazu-Tannen & Subrahmanyam, 1991). The second is that structural recursion has too much expressive power because it can express queries that require exponential time and space.

While programming languages always take Turing completeness for granted, the attitude in database programming is radically different. In the context of querying databases, due to their immense size, queries are restricted to those which are practical in the sense that they should be within 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 high probability of optimizing the query to no more than $O(n \cdot \log n)$ complexity. Thus database query languages such as SQL are designed in such a way that joins are easily recognized, as 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 (Wadler, 1992). A nested relational calculus, which is denoted here by $\mathcal{NRC}$, was then designed around this restriction (Buneman *et al.*, 1995). $\mathcal{NRC}$ 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. $\mathcal{NRC}$ (suitably extended) is implemented by the NRC Module of Kleisli and is the abstract counterpart of CPL, a la Wadler's equations relating monads and comprehensions(Wadler, 1992).

In order to show that $\mathcal{NRC}$ is a good basis for a query language, its relationship to existing query languages must be investigated. Furthermore, it has to enable

solution to existing open problems in query language theory, it has to enable generalization of existing results in query language theory, it has to facilitate practical implementation, it has to allow for good query optimization, and it has to enable new applications.

The expressive power of $\mathcal{NRC}$ and its extensions are studied in (Suciu, 1997; Dong *et al.*, 1997; Libkin & Wong, 1997; Buneman *et al.*, 1995; Suciu & Wong, 1995). These papers presented solutions to several open problems in query language theory. The most important of these results are directed at $\mathcal{NRC}(=)$ and $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. The former is $\mathcal{NRC}$ augmented with equality test. The latter is $\mathcal{NRC}(=)$ further augmented with rational numbers, linear order on rational numbers, arithmetic operations, and a summation construct. $\mathcal{NRC}(=)$ was shown to have exactly the power as the usual nested relational algebra (Buneman *et al.*, 1995). $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ was shown to capture the power of SQL, including aggregate functions and group-by constructions (Libkin & Wong, 1997). These languages are much easier to analyse than existing nested relational algebras and SQL, and thereby are likely to be easier to implement and optimize. For example, Libkin and the author(Libkin & Wong, 1997) began a series of powerful analyses on $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ that fruitfully resolved several open questions on SQL, including the following long anticipated results on unordered graphs: $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ and thus SQL cannot test if a graph is a chain, nor test if a graph is connected, nor test if a graph has an even number of edges, nor compute the transitive closure of a graph.

The impact of these and other theoretical results on the design of CPL and Kleisli is that CPL adopts $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ as its core. $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ captures all standard nested relational queries in a high-level manner that is easy for automated optimizer analysis (a primary reason that we were able to use it to prove many difficult theorems on SQL.) It is also easy to translate more user-friendly surface syntax such as the comprehension syntax or the SQL select-from-where syntax into $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. It is thus a very suitable core.

However, the fixpoint operator and several forms of structural recursion can be imported into CPL for rare occasions where recursive computations are needed. To appreciate this design decision, let us look at SQL, the de facto commercial query language. As proved in (Libkin & Wong, 1997), SQL cannot express recursive queries. In practice, although fixpoint computations are sometime desirable, it is of less importance in the database context than counting. Indeed, in SQL, a limited recursion construct has only been recently proposed and is still being debated for the latest standard, while aggregate functions such as summation and average have always been part of SQL (they belong to the so-called "entry-level" SQL92, which is supported by all commercial relational database systems).

3 Impact on Bioinformatics

As mentioned earlier, the Kleisli system was developed as a general solution to broad-scale data integration problems. The author and his collaborators used the

bioinformatics arena as the first testbed of the old 1994 prototype of the system (Davidson *et al.*, 1997) and succeeded in making an impression on that field (Leser *et al.*, 1998; Benton, 1996; Karp, 1996; Baker & Brass, 1998). A brief commentary to the functional programming community on the success of Kleisli was also given by Wadler (1998). The description of Kleisli in action in bioinformatics provided by these previous papers was perhaps overly simplified. So we provide a detailed real-life example in querying protein patents for illustration. The challenges of data integration in bioinformatics is explained in Subsection 3.1. The motivation behind the detailed example and the protocol involved is explained in Subsection 3.2. The CPL query implementing the example is exhibited and explained in Subsection 3.3.

3.1 Data Integration

"Until recently, biological sequence databases were built by biologists. When sequence databases were first created the amount of data was small and it was important that the database entries were human readable. Database entries were constructed, therefore, as flat files, that is, text entries with the information ordered in a specific way. Indeed, it is probably more accurate to describe these databases as data repositories. As new types of data were captured or created, new data repositories were created using a variety of flat file formats. The result of this effort has been to create a large number of different databases, all in different formats, typically using non-standard data query software, and only really properly accessible to bioinformatics experts" (Baker & Brass, 1998).

It is a significant challenge if these pieces have to be used together in complex ways to answer new questions in biology. Clearly, simple retrieval of data is not sufficient for modern bioinformatics. The challenge is how to manipulate the retrieved data derived from various databases and re-structure the data in such a way to investigate specific biological problems. This may require feeding the retrieved data into various application programs, such as multiple sequence alignment programs, 3D structure modeling programs, and so on, which require specific input data sets and formats.

As observed by Baker and Brass (Baker & Brass, 1998), many existing biology data retrieval systems are not fully up to the demand of flexible and painless data integration. These systems rely on low-level direct manipulation by biologists. The archetypal example is the Entrez system (Schuler *et al.*, 1996). Here a biologist uses a keyword to extract summary records, then click on each record to view its contents or perform operations. This works well for simple actions. However, as the number of actions or records increases, such direct manipulation quickly becomes a repetitive drudgery. Also, when the questions become more complex and involve many databanks, assembly of the data needed exceeds the skill and patience of most biologists. Merely providing a library package that interfaces to a lot of databases and analysis softwares is also not useful if it requires long-winded and tedious programming to make use of and/or adding to the package, as demonstrated (Selletin & Mitschang, 1998) by the difficulties with CORBA (Siegel, 1997).

As described earlier, Kleisli (Davidson *et al.*, 1997) goes one step further and provides the high-level and powerful query language CPL. CPL offers a nice data

model and many high-level operators to express complex queries and tranformations on these biology databases and analysis softwares in a manner that is extremely straightforward and without overly taxing a user's programming skill. It is now widely agreed that Kleisli has significantly reduced the difficulty of integrating biology data (Benton, 1996; Karp, 1996; Baker & Brass, 1998). To get a sense of Kleisli's impact on bioinformatics, let us describe the very first bioinformatics query implemented in Kleisli in 1994 (Davidson *et al.*, 1997).

It was one of the so-called "impossible" queries of a US Department of Energy Bioinformatics Summit Report.* That 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, this query means that for every gene found on 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 (Pearson *et al.*, 1992), 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 (Burks *et al.*, 1992). At the time, access to GenBank was provided through the ASN.1 version of Entrez (Schuler *et al.*, 1996), which was an extremely complicated retrieval system. Entrez also kept precomputed homologs of GenBank sequences.

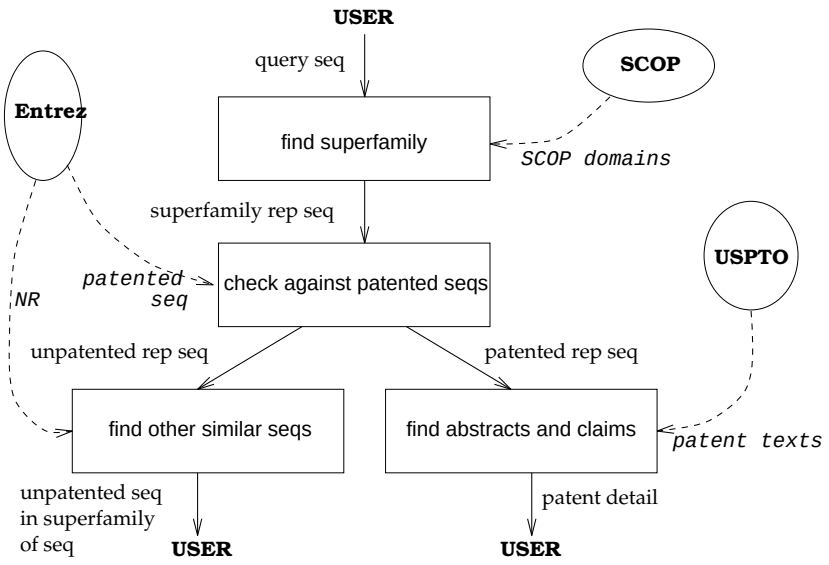
So this query required the integration of GDB (a relational database) and Entrez (a non-relational "database") that first extracted names of genes on the required cytogenetic band and accessed Entrez for the homologs of these genes and finally filtered these homologs to retain only the non-human ones. This query was considered "impossible" because 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 CPL query given in (Davidson *et al.*, 1997) was remarkably short.

Since then Kleisli has been used to power many bioinformatics applications (Baker *et al.*, 1998; Chen *et al.*, 1998a; Chen *et al.*, 1998b; Chen *et al.*, 1998c). In the next two subsections, a bioinformatics application in protein patents that is easily understood by non-biologists is described.

3.2 Querying Protein Patents

Consider a pharmaceutical company that has a large choice of protein sequences to work on (ie. to determine their functions.) A criterion for selection is patent potential. A process involving many data sources and steps is required, as depicted in the "dataflow" diagram below.

* `http://www.gdb.org/Dan/DOE/whitepaper/contents.html`



At the initial stage, only the amino acid sequence of these sequences and very little else is known. A question at this point would be: Which of them have already been patented? Existing patent search systems are information-retrieval systems that rely on English words. These systems suffer from the dichotomy of recall vs precision (Frakes & Baeza-Yates, 1992). They either return only the highly relevant information at the expense of missing out a large proportion of them, or return most of the highly relevant information at the expense of returning also a lot of irrelevent information. So searching patents using them is laborious and is not always fruitful. Furthermore, at this early search, we do not even have any English word to use for searching—we have only the actual amino acid sequences! Thus, more reliable technology for comparing these protein sequences to those sequences already patented is needed.

There are two things that come to the rescue. First, protein patents are generally based on protein function *and* primary sequence structure (ie. the linear string of 20 symbols in the amino sequence.) Second, bioinformatics is sufficiently advanced and tools that can reliably identify homology at the primary sequence structure level are available (Pearson, 1995). Therefore, if patented protein sequences can be extracted from some database and prepared in a form suitable for such tools to operate on, there is a means to reliably identify which of the company's sequences have not yet been patented. The patented sequences can be obtained from the protein section of Entrez (Schuler *et al.*, 1996). These are "warehoused" locally for greater efficiency. The sequence comparison software, WU-BLAST2 (Altschul & Gish, 1996), can be used for comparing the company's sequences against this warehouse for primary sequence structure homology.

After the unpatented protein sequences have been identified, the second question at this point is: Which ones of these have the potential for wider patent claims? To understand this question it is necessary to recognize that protein patents are generally granted on a sequence *and* its function. While the functions of the company's

proteins are unknown at this point, because no work has been done on them yet, it is known that proteins of the same evolutionary origin tend to have similar functions even if they differ significantly in their primary structure (Lodish *et al.*, 1995). Using the terminology of structural classification of protein database of (Murzin *et al.*, 1995) (officially known as *scop*, in lower case), these proteins are in the same superfamily. So one way to identify protein sequences that have potential for wider patent claims is to find those having large number of unpatented sequences in the same superfamilies. Homology searching algorithms based on primary structure are generally not sufficiently sensitive to detect the majority of sequences in a typical superfamily (Pearson, 1995), as the primary structure of distance members of the family are likely to have mutated significantly. So tools for homology search at the tertiary structure level are needed.

No reliable automatic tools for this purpose exists at this moment, because structural similarity at the tertiary level does not necessarily imply similarity in function. Nevertheless, reliable manually constructed databases of superfamilies exist. A very nice one is scop (Murzin *et al.*, 1995). Therefore, if the unpatented sequences can be screened against scop, it is possible to pull out other *representative* sequences in their superfamilies and check which ones have already been patented, thus identifying superfamilies with good potential. WU-BLAST2 is a reliable sequence comparison program suitable for carrying out the screening. After unpatented representatives of superfamilies have been found, it is still necessary to use them to fish out the rest of the unpatented members of superfamilies. This step can be accomplished by using Gapped BLAST (Altschul *et al.*, 1997) to remotely compare these representatives against the huge nonredundant protein database (NR) curated at the National Center for Biotechnology Information.

Having found these potentially good protein sequences, the pharmaceutical company is ready to work on them and to hopefully patent the results in due course. The company is ready to ask the third question: What are the relevant prior arts? The examination of prior arts is also necessary to determine directions when the company begins work on their proteins, if it decides to work on them at all. Retrieving the texts of patented sequences in the same superfamilies as their proteins would be very helpful here. This step is complementary to the previous step and can be carried out using exactly the same technology! In the next subsection, the CPL program that accomplishes this step is presented.

3.3 Patented Proteins in a Superfamily

This subsection concentrates on explaining this example query:

Example 3.1
The CPL program below answers this question: *Find the superfamily that a given protein sequence* `SEQ` *belongs to, and identify patented members of this superfamily.*

```
1. localblast-blastp (#name: "scop-blast", #db: "scopseq");
2. localblast-blastp (#name: "pat-blast", #db: "patseq");
3. scop-add "scop";
4. setindex-access (#name:"sid2seq", #file: "scopseq", #key: "#sid");
```

```
 5. {(#sf: (#desc: xinfo.#desc.#sf, #hit:x.#accession, #pscore:x.#pscore),
 6.   #bridge: (#hit: s, #patent: p.#title, #pscore: p.#pscore))
 7. | \x <- process SEQ using scop-blast, x.#pscore <= PSCORE,
 8.   \xinfo <- process <#sidinfo: x.#accession> using scop,
 9.   \s <- process <#numsid: xinfo.#type.#sf> using scop,
10.   \y <- process <#key: s> using sid2seq,
11.   \p <- process y.#seq using pat-blast, p.#pscore <= PSCORE };
```

□

This query requires the following data sources: scopseq, which contains the representative sequences for various superfamilies in scop; patseq, which contains the patented protein sequences extracted from Entrez; and scop (Murzin *et al.*, 1995), which contains the structural classification of proteins. Each entry in scopseq is uniquely identified by a name called *sid*, on which an index is available. The scop database is a specialized deeply nested database and comes with a specialized retrieval software that supports four commands, two of which are used here: given a *sid*, return the corresponding scop entry; given a scop classification number, return the *sid* of all scop entries belonging to that class. This query also requires WU-BLAST2, which is a good sequence comparison software. Both scopseq and patseq have already been "warehoused" by the Kleisli system before hand in a format compatible with WU-BLAST2 (Altschul & Gish, 1996) for greater efficiency. So we have a query that requires three different sources and two different softwares that are not typical database systems.

A connection `scop-blast` is established between WU-BLAST2 and scopseq (line 1). A connection `pat-blast` is established between WU-BLAST2 and patseq (line 2). These connections allow WU-BLAST2 to compare a given input protein sequence against the sequences in scopseq and patseq to identify those that are homologous or highly similar to the input protein sequence. A connection `scop` is establish on scop (line 3). This connection provides classification information on proteins. To provide random access to representative sequences of scop superfamilies given their scop identifier *sid*, a connection `sid2seq` is established on the *sid* index of scopseq (line 4). Having set all these connections up, the real work can start.

The given sequence `SEQ` is compared for sequence similarity against scop superfamily representative sequences in scopseq using `scop-blast` to look for a hit `x` at a strong given threshold `PSCORE` (line 7). `SEQ` is regarded to be in the same superfamily as such a hit `x`. The *sid* of `x` is recorded in the `#accession` field of `x`. The protein classification record `xinfo` of `x` is looked up from scop using `scop` and the *sid* of `x` (line 8). This record has a field `#type`, which is another record. This inner record has a field `#sf`, which stores the scop classification number of `x`'s superfamily. Using this classification number and the `scop` connection, we ask scop for all the *sid* `s` of representatives in the superfamily of `x` (line 9). Each representative `s` serves as a "bridge" to help us identify the desired patented sequences. For each such representative `s`, we use the index `sid2seq` to look up its sequence record `y` (line 10). The `#seq` field of this record stores the protein sequence of `y`. This sequence is then "BLASTed" against patented sequences in patseq using the connection `pat-blast`; those hits `p` at the given strong threshold `PSCORE` are the patented sequences desired

(line 11). The relevant information (the superfamily information of `x`, the bridge `s`, and the patent information `p`) are returned in the output set (lines 5–6).

4 Influence of Functional Programming

Functional programming has a significant influence on the design and implementation of Kleisli and CPL. This influence is most visible at the language level of CPL. CPL has higher-order functions and a ML-style polymorphic type system. Its type system is further augmented with another invention from the functional programming community: parametric record polymorphism (Ohori *et al.*, 1989; Remy, 1989). Although CPL's view of record type variables is closer to that in Machiavelli (Ohori *et al.*, 1989), than the row variables of Remy (1989), the implementation (Wong, 1995a) of polymorphic records in Kleisli is based on a clever idea of Remy (1992). The most noticeable feature of CPL, the comprehension syntax, made its appearance many years ago in the programming language world in languages such as Miranda. It was also discussed earlier and elsewhere (Buneman *et al.*, 1995) that the core of CPL is founded on structural recursion (Breazu-Tannen *et al.*, 1991) and monad (Wadler, 1992).

It is more interesting to discuss the influence of functional programming on components of the Kleisli system that are less visible. In particular, we discuss the self-describing exchange format of Kleisli in Subsection 4.1, the abstract internal representation of queries in Subsection 4.2, and the optimizer in Subsection 4.3.

4.1 Type Inference and Self-Describing Exchange Format

Of the many discoveries by the functional programming community, our favourite is parametric polymorphism and type inference. CPL uses such a type system and Kleisli's self-describing data exchange format is also a direct derivative of such a type system. The benefits are discussed in this subsection.

In a dynamic heterogeneous environment such as that of bioinformatics, many databases and softwares are used. Worse still, they often do not have any thing that can be thought of as an explicit database schema. Further compounding the probem is that research biologists demand flexible access and queries in very 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 completely on the structure of that query. Second, it must have a data model that these external databases and softwares can easily translate to, without doing a lot of type declarations and so on. Third, it must shield existing queries from evolution of these external databases and softwares as much as possible. For example, an extra field appearing in an external database table must not make it necessary to recompile/rewrite an existing query. 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 from external databases and softwares.

The first three of these requirements are addressed directly by several interesting features of CPL's type system. CPL has polymorphic record types that allow, for example, `\R => { x.#name | \x <- R, x.#salary > 1000}`, which defines a function that returns names of people in `R` earning more than a thousand dollars. This function can be applied to any `R` that has at least the `#name` and the `#salary` fields, thus allowing the input source some freedom to evolve. CPL also has variant types that allow, for example, `{ <#name: "John">, <#zip-code: 119613> }`, which is a set containing objects of very different structures; in this case, a string carrying a `#name` tag and a number carrying a `#zip-code` tag. This feature is particularly useful in handling ASN.1-formatted (ISO, 1987) data from GenBank (Burks *et al.*, 1992), one of the most important and most complex sources of DNA sequences, as it contains a profusion of variant types.

Note that functional programming languages like Haskell and SML require variant types to be declared in advance, and Haskell does not even have first class record types. In contrast, CPL does not require variant types to be declared at all. In fact, CPL does not require any type to be declared at all. Of course, the type and meaning of any CPL 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 big explicit schemas that run into tens of pages—an example big complex schema is the ASN.1 schema of Entrez (National Center for Biotechnology Information, 1992)—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 value construction in CPL, as described in Subsection 2.2. Recall that CPL programs contain no type declaration. A CPL compiler has to figure out if a CPL program has a principle typing scheme. This kind of type inference is possible because every construct in CPL has an unambiguous most general type. In particular, the value construction syntax is such that it is possible to inspect only the first several symbols to figure out local type constraints on the corresponding value, as each value constructor is unambiguous. For example, if a `{|` bracket is seen, it is immediately clear that it is a bag; and if a `(` bracket is seen, it is immediately clear that it is a record. Thus, by adopting the value construction syntax of CPL as the data exchange format, the latter becomes self describing.

A self-describing exchange format is one in which there is no need to define in advance the structure of the objects being exchanged. In database terminology, it means there is no fixed schema. In programming language terminology, it

means there is 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 (ISO, 1987) 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 a simple one.

It should be mentioned that self-describing data exchange formats exist in several forms in earlier work (Papakonstantinou *et al.*, 1995; Litwin *et al.*, 1990). However, Kleisli's is probably the first self-describing exchange format that is consciously derived from type inference! It should be noted that Kleisli also has a self-describing *compressed* data exchange format that is more compact than the value construction syntax of CPL. For example, instead of `{(#name: "John", #age: 10), (#age:12, #name: "Peter"), ...}`, one uses the shorter `{|DEFR1 #name #age ("John" 10), |R1 ("Peter", 12), ...}`. In the former, the fields of records are position independent, at the cost of having to repeat record labels many times. In the latter, the record labels are declared in advanced and the fields of records are fixed according to the order of declaration of their corresponding labels, so that it is not necessary to repeat record labels. Essentially, `|DEFR`*i* works like a data type declaration defining the "constructor" `|R`*i*, whose scope is restricted to the particular data stream from the point that particular `|DEFR`*i* appears to the end of that data stream or to the point `|DEFR`*i* is redefined in that data stream, whichever comes earlier. Thus the simple compressed exchange format essentially retains the flavour imparted by CPL's value construction syntax.

4.2 Kleisli Triples and Abstract Syntax

Let us briefly recall the restricted form of structural recursion which corresponds to the presentation of monads by Kleisli (Wadler, 1992; Buneman *et al.*, 1995). It is the combinator $ext(\cdot)(\cdot)$ obeying these three equations: $ext(f)\{\} = \{\}$, $ext(f)\{o\} = f(o)$, and $ext(f)(A \cup B) = ext(f)(A) \cup ext(f)(B)$. Thus, $ext(f)(R)$ is equal to the $\bigcup\{f(x) \mid x \in R\}$ construct of $\mathcal{NRC}$. The direct correspondence in CPL is: `ext{`e_1 `| \x <-` e_2 `}`, which is interpreted as $ext(f)(e_2)$, where $f(x) = e_1$. This combinator is a key operator in the Complex Object Library of Kleisli and is at the heart of the NRC, the abstract representation of queries in the implementation of CPL. It earns its central position in the Kleisli system because it offers tremendous practical and theoretical convenience.

Its practical convenience is best seen in the issue of abstract syntax in the implementation of a database query language. The abstract syntax is the internal representation of a query and is usually manipulated by code generators; the better abstract synax is the one that is easier to analyse. It must not be confused with the surface syntax, which is what the usual database programmer programs in; the better surface syntax is the one that is easier to read. It is worth contrasting the `ext` construct to the comprehension synax here. With regard to surface

syntax, CPL adopts the comprehension syntax because it is a easier to read than the `ext` construct. For example, the Cartesian product of two sets is expressed using the comprehension syntax as `{(x, y) | \x <- R, \y <- S}`. In contrast, it is expressed using the `ext` construct as `ext{ext{{(x,y)} | \y <- S} | \x <- R}`, which is more convoluted. However, the advantage of the comprehension syntax more or less ends here. With regard to abstract syntax, the situation is exactly the opposite! Comprehensions are easy for the human programmer to read and understand. However, they are in fact extremely inconvenient for automatic analysis and is thus a poor candidate as an abstract representation of queries. This difference is illustrated below by a pair of contrasting examples in implementing optimization rules.

A well-known optimization rule is vertical loop fusion (Goldberg & Paige, 1984), which corresponds to the idea of getting rid of intermediate data. Such an optimization on queries in the comprehension syntax can be expressed informally as

$$\begin{array}{l} \quad \{e \mid G_1, ..., G_n, \backslash x \texttt{ <- } \{e' \mid H_1, ..., H_m\}, J_1, ..., J_k\} \\ \rightsquigarrow \{e[e'/x] \mid G_1, ..., G_n, H_1, ..., H_m, J_1[e'/x], ..., J_k[e'/x]\} \end{array}$$

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 $\backslash x$ `<-` $\{e' \mid H_1, ..., H_m\}$ and J_i. The author tried to find a simple technique to implement such a rule for over three years but did not succeed.

Let us now consider the `ext` construct. As pointed out by Wadler (1992), any comprehension can be translated into this contruct. Its effect on the optimization rule for vertical loop fusion is dramatic. This optimization is now expressed as

$$\begin{array}{l} \quad \texttt{ext}\{e_1 \mid \backslash x \texttt{ <- ext}\{e_2 \mid \backslash y \texttt{ <- } e_3\}\} \\ \rightsquigarrow \texttt{ext}\{\texttt{ ext}\{e_1 \mid \backslash x \texttt{ <- } e_2\} \mid \backslash y \texttt{ <- } e_3\} \end{array}$$

The informal and troublesome "..." no longer appears. Such a rule can be coded up straightforwardly in almost any implementation language. A similar simplication is also observed in proofs using structural induction. For comprehension syntax, when one comes to the case for comprehension, one must introduce a secondary induction proof based on the number of generators and filters in the comprehension, whereas the `ext` construct does not give rise to such complication. A related saving is that comprehensions require two kinds of terms, expressions and qualifiers, whereas the `ext` formulation requires only one kind of terms, expressions.

In order to illustrate this point more concretely, it is necessary to introduce some detail from the implementation of the Kleisli system (Wong, 1995b). The type `SYN` of ML objects that represent queries in Kleisli is declared in the NRC Module mentioned in Subsection 2.1. The data constructors that are relevant to our discussion are given below, with the corresponding CPL constructs enclosed within ML comment brackets. It is some times convenient to think of an object of type `SYN` as an expression tree. For example, `IfThenElse`(e_1, e_2, e_3) can be thought of as a tree, with e_1, e_2, and e_3 as subtrees, and `IfThenElse` as root.

```
type VAR = int            (* Variables, represented by int *)
type SVR = int   (* Server connections, represented by int *)
datatype SYN
= ...
| EmptySet                        (* { }                    *)
| SngSet of SYN                   (* { E }                  *)
| UnionSet of SYN * SYN           (* E1 {+} E2              *)
| ExtSet of SYN * VAR * SYN       (* ext{ 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 *)
```

All ML objects that represent optimization rules in Kleisli are functions and they have type `RULE` as given below.

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

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

We now return to the optimization rule on vertical loop fusion. As promised earlier, we are rewarded by a simple implementation:

Example 4.1
Vertical loop fusion.

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

□

Many other rules that have unpleasant implementation based on the comprehension syntax also become straightforward under the Kleisli monad representation. A second one is presented here for illustration. It is easy to see what this rule means:

{e | ..., \x <- G, C, ...}
$\leadsto$ {e | ..., C, \x <- G, ...}, provided x is not free in C.

This rule is called "filter promotion" in the functional programming world. It is a simple form of code motion (Aho *et al.*, 1986) that migrates a filter C out of an inner loop (the iteration of x over every element of G), which achieves the dual purpose of avoiding recomputing C multiple times and—should C be false—avoiding the computation of G and all the inner loops due to the iteration of x over G. Again the informal "..." presents a major challenge to implement this rule in a simple way. However, this rule has a simple correspondence (that is even slightly more general) when expressed using the `ext` construct:

`ext{ if` C `then` e_1 `else` e_2 | \x <- e_3}
$\leadsto$ `if` C `then ext{`e_1 | \$x$ <- e_3`} else ext{`e_2 | \$x$ <- e_3}, provided x is not free in C.

Not surprisingly, its realization in Kleisli is pleasantly simple:

Example 4.2
Filter promotion.

```
fun SimpleCodeMotion(ExtSet(IfThenElse(C, E1, E2), x, E3))
= if IsFree x C
  then NONE
  else SOME(IfThenElse(C, ExtSet(E1, x, E3), ExtSet(E2, x, E3)))
| SimpleCodeMotion _ = NONE
```

□

The surface syntax of CPL is based on comprehension syntax while its internal abstract representation is based on the `ext` construct inspired by the Kleisli monad. The former is a good choice for implementing *in* CPL. The latter is a good choice for implementing CPL. Fortunately, it is also well known that an easy translation exists between them (Wadler, 1992; Buneman *et al.*, 1995). The translations are given by the following "equations":

$\{e_1 \mid \backslash x$ `<-` $e_2, ...\} =$ `ext`$\{\{e_1 \mid ...\} \mid \backslash x$ `<-` $e_2\}$

$\{e \mid \} = \{e\}$

$\{e \mid C, ...\} =$ `if` C `then` $\{e \mid ...\}$ `else` $\{\}$

Thus, it is possible to have a pie and eat it too!

The Kleisli optimizer also performs many other optimizations. These optimizations include a more general form of code motion; parallelism to exploit network latency; selective introduction of laziness to reduce memory consumption and to improve response time; migration of selection, projection, and joins to external relational database servers; reordering of joins across tables from distinct database servers; etc. Some of these were described in (Davidson *et al.*, 1997). The impact of some of these optimizations are discussed in Section 5.

4.3 Higher-Order Functions and Optimization

There is another very pleasant experience in implementing the optimizer for the Kleisli system that illustrates very well the many advantages and conveniences of higher-order functions, besides allowing the expression of better algorithms as discussed in (Suciu & Wong, 1995). The optimizer (Wong, 1995c) consists of 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. The more familiar include `BottomUpOnce`, which applies rules to rewrite an expression tree from leaves to root in a single pass; `TopDownOnce`, which applies rules to rewrite an expression tree from root to leaves in a single pass; `MaxOnce`, which applies rules to the largest redices in a single pass; and so on, together with their multi-pass versions.

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. In order to give some ideas on how this is done, some code fragments from the optimizer module mentioned in Subsection 2.1 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 ML objects that represent 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 f (SngSet N) = SngSet (f N)
|   Decompose f (UnionSet (N, M)) = UnionSet (f N, f M)
|   Decompose f (ExtSet (N, x, M)) = ExtSet (f N, x, f 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 ML 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 a very simple way.

Example 4.3
The `MaxOnce` strategy applies rules to maximal subtrees. It starts trying the rules on the root of the query expression. If no rule can be applied, it moves down one level along all paths and tries again. But as soon as a rule can be applied along a path, it stops at that level for that path. In other words, it applies each rule at most once along each path from the root to the leaves. Here is its "control" component:

```
fun MaxOnce RDB Query =
  case Pick RDB Query
  of SOME ImprovedQuery => ImprovedQuery
  | NONE => Decompose (MaxOnce RDB) Query
```

□

Example 4.4
The `TopDownOnce` strategy applies rules from the root down to the leaves. It tries to rewrite each node of the query expression at most once as it progresses to the leaves. Here is its "control" component:

```
fun TopDownOnce RDB Query =
  let fun Pass Subquery =
    let val ImprovedSubquery =
      case Pick RDB Subquery
      of SOME ImprovedSubquery => ImprovedSubquery
      | NONE => Subquery
    in Decompose Pass ImprovedSubquery end
  in Pass Query end
```

□

Example 4.5
The `BottomUpOnce` strategy applies rules from the leaves to the root. 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 Query =
  let fun Pass Subquery =
    let val ImprovedSubquery = Decompose Pass Subquery
    in case Pick RDB ImprovedSubquery
       of SOME FurtherImprovedSubquery => FurtherImprovedSubquery
       | NONE => ImprovedSubquery end
  in Pass Query end
```

□

Let us now present an interesting class of rules that requires the use of multiple rule application strategies. The scope of the rules (vertical loop fusion and filter promotion) in the previous subsection 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 component 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 a ML 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 transformation 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 4.6

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 ThenBranch E =
         if SyntaxTools.Equiv E1 E then SOME(Bool true) else NONE
   in case ContextSensitive ThenBranch 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 `ThenBranch` 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 Query =
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 OptimizedQuery = Strategy RDB Query  (* Apply Rule using Strategy. *)
in if !Changed then SOME OptimizedQuery else NONE
end
```

This `ContextSensitive` rule generator is reused for many other context-sensitive optimization rules, such as those used for migrating selection, projections, and joins to external relational database systems. □

Thus the use of higher-order functions greatly simplifies the implementation of the current Kleisli optimizer (Wong, 1995c), compared the original optimizer from (Wong, 1994). It should be mentioned that the author is *not* the first to discover this particular method of implementing rewrite strategies; Paulson (1983) and Spivey (1990) presented similar ideas before.

5 Optimization and Performance

Many optimizations emphasized in Kleisli are not traditionally studied in functional programming languages or in database management systems. This section uses an example query that joins two relational databases to give an indication of the kind of optimizations that Kleisli performs on queries involving external sources and

their impact. The idea of migrating joins to data sources is used to improve overall performance in Subsection 5.2, the idea of lazy evaluation is used to reduce response time in Subsection 5.3, and the idea of concurrency is used to reduce network idle time in Subsection 5.4.

5.1 Original Query

The unoptimized CPL query below accesses two relational databases in the east coast of the United States and joins their data. The query was submitted to a Kleisli/CPL site in Singapore in 1995. There was then a conjested T1 internet link between Singapore and the United States.

Example 5.1
Sortez (Hart *et al.*, 1994a) is a relational database containing certain information extracted periodically from Entrez. GDB (Pearson *et al.*, 1992) is a relational database containing mapping information. We would like to use GDB to obtain the first 20 records that fall into Human Chromosome 22. Then for each of these records, retrieve its related records in Sortez. Assuming that some connection(s) `gdb` has already been established to GDB and some connection(s) `sortez` has already been established to Sortez, this task can be accomplished by the query `combine` as follows:

```
primitive loci22 == set-firstn (20,
 {(#genbank_ref: x.#genbank_ref, #locus_symbol: y.#locus_symbol)
 | \x <- process "select * from object_genbank_eref g where 1=1" using gdb,
   x.#object_class_key = 1,
   \y <- process "select * from locus l where 1=1" using gdb,
   y.#locus_id = x.#object_id,
   \z <- process "select * from locus_cyto_location c where 1=1" using gdb,
   z.#locus_id  = x.#object_id, z.#chrom_num = "22" }) ;

primitive combine ==
 {(#gdb: x,
   #sortez: {(w.#locus, w.#accn, w.#title, w.#len, w.#taxname)
      | \w <- process "select * from gb_head_accs g where 1=1" using sortez,
        w.#pastaccn = x.#genbank_ref })
  | \x <- loci22};
```

□

The `process` M `using` S construct works by sending the request M to the specified connection S and then parsing the corresponding reply into a complex object. Kleisli supports multiple connections to all of its data servers. For the discussion in this section, we use one connection for GDB (any request sent down `gdb` is directed to this connection) and two for Sortez (any request sent down `sortez` is directed to one of these connections, whichever is currently free.)

As optimization concerns evaluation, we need to describe the evaluation behaviour of the comprehension construct. For sets, this construct usually has the form { E | P_1 <- E_1, ..., P_n <- E_n, C_1, ..., C_m}. E, E_i, and C_j can be arbitrary expressions. However, E_i must evaluate to sets and C_j must evaluate to Booleans. Each P_i is a pattern used for matching elements in the corresponding set E_i. This

construct is evaluated in the absence of optimization using a nested loop. For example, when $n = 2$, the evaluation algorithm looks like this:

```
initialize accumulator to emptyset
foreach matching P1 in E1
do foreach matching P2 in E2
   do if C1 and ··· and Cm
      then insert E into accumulator
   end
end
return accumulator
```

The `set-firstn` function simply selects the "first" n items in a set; in our example above, it selects the first 20 records about the locus of sequences on chromosome 22. We obtained the timings at the end of this section by increasing this number.

Therefore, if we ran `loci22`, this would cause GDB to be accessed repeatedly to compute the three-way join of the three tables `object_genbank_eref`, `locus`, and `locus_cyto_location` of sizes approximately 150000, 80000, and 80000 records respectively. There would be one access to bring in the `object_genbank_eref` table. Then there would be n accesses to bring in the `locus` table n number of times, with n being the cardinality of the `object_genbank_eref` table. Then there would be $n \cdot m$ accesses to bring in the `locus_cyto_location` table $n \cdot m$ times, with m being the cardinality of the `locus` table.

Consequently, if we ran `combine`, this would cause both GDB and Sortez to be access many times to compute the join between `loci22` and the `gb_head_accs` table. There would first be $1 + n + n \cdot m$ accesses to GDB to compute `loci22` as described above. Then there would be an additional 20 accesses to Sortez to compute the join between the `gb_head_accs` table and the first 20 records in `loci22`.

It is obvious that such a query would take a long time to produce any output, even if we had an unconjested high-capacity T3 link between Singapore and the United States, let alone the conjested T1 link in 1995. It is clearly unacceptable. In the following subsections, we describe some optimizations that Kleisli carries out automatically.

5.2 Maximal Subquery Migration

The obvious optimization is to migrate operations to data sources whenever possible. The rationale for this optimization is that certain statistics and indices are available to the native optimizers at some of these data sources. Therefore, they can be expected to perform the operations more efficiently. In fact, we should migrate subqueries that are as large as possible (Ngu *et al.*, 1993) to execute remotely at the data sources. The effect of migrating maximal subqueries is to ensure that the native optimizers have as much context as possible and to give them as much work as possible.

Example 5.2

The equivalent query after the migration of the three-way join needed to compute `loci22` to GDB is shown below.

```
primitive loci22' == set-firstn (20,
 process "select genbank_ref = g.genbank_ref, locus_symbol = l.locus_symbol
          from locus l, locus_cyto_location c, object_genbank_eref g
          where l.locus_id = g.object_id and g.object_id = c.locus_id
          and g.object_class_key = 1 and c.chrom_num = '22'"
 using gdb);

primitive combine' ==
 {(#gdb: x,
   #sortez: {(w.#locus, w.#accn, w.#title, w.#len, w.#taxname)
       | \w <- process "select locus = g.locus, accn = g.accn, len = g.len,
                               title = g.title, taxname = g.taxname
                        from gb_head_accs g
                        where g.pastaccn = " ^ sql-string(x.#genbank_ref)}
             using sortez)
 | \x <- loci22'};
```

□

`loci22'` sent just one SQL request to GDB to obtain the desired three-way join. So executing `combine'` required merely one access to GDB and 20 accesses to Sortez.

Furthermore, each of the 20 accesses to Sortez in `combine'` was much cheaper than that in `combine`. The reason being that `combine` brought in the `gb_head_accs` table completely each time, whereas `combine'` brought in only a few columns of that table (because the projections on these columns had been migrated) and only a few rows of that table (because the selection on the `pastaccn` column had been migrated).

This was a big improvement, for the query was completed in about 42 seconds. However, in this version of the query, the first record was not displayed until the entire result table was computed. Hence, its response time was also around 42 seconds. If we were joining more records than the first 20, we would have to wait a rather long time to see anything.

5.3 Lazy Evaluation

To improve response time, we should start the output process as soon as some rows in the resulting table become available. This can be accomplished by lazy evaluation. The idea of lazy evaluation is implemented by changing `combined'` into a query `combined''` by replacing the outermost `{`-bracket with a `{l`-bracket. The presence of a `l`-tag in a comprehension construct indicates that the result of the comprehension is to be formed lazily. That is, as soon as some elements of the set become available, the system should return them, without waiting for the rest of the elements in the set. The rest of the elements are returned on demand when they become available. CPL takes care of coordination automatically.

Thus while `combine''` required the same number of accesses to GDB and Sortez as `combine'`, it started displaying the first row of its result after one access to

GDB and one access to Sortez. The response time was now about 7 seconds, greatly reduced from the 42 seconds of `combine'`. However, it still needed about 42 seconds to complete the query.

5.4 Concurrent Evaluation

Sortez was accessed once for each item in `loci22'`. The accesses to Sortez were essentially sequential in the previous versions of our example. Rather than sequentially sending requests to Sortez, we should be able to exploit the fact that many data servers (such as Sortez which ran on top of Sybase) could handle several requests simultaneously. Similarly, while our system was waiting for a response from Sortez, it had enough resources to send a new request to Sortez or any other servers and to process the reply to its previous request simultaneously. Since Kleisli/CPL has built-in capability to coordinate multiple threads of execution automatically, we should take advantage of it to overlap all these waiting times.

The idea of using concurrency to reduce waiting is implemented by changing `combine''` into a new query `combine'''` by replacing the outermost generator `\x <- loci22'` by `\x <<- loci22'`. In CPL, P `<-` E means matching the pattern P over elements of the set E in a sequential fashion. On the other hand, P `<<-` E means matching the pattern P over elements of the set E in parallel. Thus `combine'''` took about 10 seconds to display the first row and finished the last row in about 19 seconds, using two connections to the `sortez` server concurrently.

In introducing such concurrency, we must be careful of two things. First, the remote server may only be able to handle a limited number of requests at a time, say five. In this case, we should send five requests at a time to avoid overwhelming the remote server. Fortunately, data servers in our application are usually running on top of commercial systems like Sybase which practice admission control. In addition, Kleisli provides mechanism to adjust its level of concurrency with respect to different servers. Second, each concurrent thread requires resources such as memory to be allocated if their output are not consumed quickly enough. This is tougher to deal with than the first problem. Kleisli relies on a priority-based scheme to schedule its external data requests and it has an optimization rule for assigning priority.

5.5 Optimization Results at a Glance

We tabulate below the typical execution time with respect to the number of `loci22` entries. All results were obtained in 1995 by running our queries on a SPARC 20 with 32 MB memory in Singapore. The data sources were located in the east coast of the United States in Philadelphia (for Sortez) and Baltimore (for GDB). All accesses relied on a conjested T1 link between the two countries. Note that for `combine'''` we used two connections to Sortez.

		Number of Entries					
		20	50	80	110	140	170
Total	`combine'`	42	53	77	109	133	179
Time	`combine''`	26	63	91	142	142	159
(sec)	`combine'''`	19	37	49	64	80	100
Response	`combine'`	42	53	77	108	132	178
Time	`combine''`	7	11	8	7	9	13
(sec)	`combine'''`	10	9	8	7	8	14

The performance improvement shown above had been achieved using CPL without complicated programming. In fact, Kleisli's optimizer can automatically perform all of the optimizations discussed above, as well as many other optimizations.

Note that if write permission were granted on Sortez, the alternative optimization of semi-join (Ullman, 1989) could be used to achieve potentially greater improvement than the concurrent version of `combine'''` above. However, the concurrent idea of `combine'''` is a more general form of optimization than the semi-join idea. In particular, the semi-join idea works well only if the data source is a powerful system such as a full-fledged relational database.

6 Evolution of Kleisli

Kleisli has undergone extensive re-design and re-implementation at the Kent Ridge Digital Labs since the original prototype built in 1994 at the University of Pennsylvania. This section has two purposes: To show how Kleisli has changed and to explain advanced features of the current release of the system (Kleisli Version D Release 21 December 1998). We focus here on granularity of laziness (Subsection 6.1), exploitation of concurrency (Subsection 6.2), improvement to the type system (Subsection 6.3), improvement to the optimizer (Subsection 6.4), availability of new drivers (Subsection 6.5), and development of new high-level query interfaces (Subsection 6.6).

6.1 Laziness

The old 1994 prototype did have laziness. Unfortunately, its implementation of laziness contained a bad idea. The bad idea was the mistaken belief that the granularity of laziness should be as fine as possible. As a result, the lazy representation of a complex object such as `{{1, 2}, {3, 4, 5}}` in the 1994 old prototype was a function f_1 that when applied to the unit object `()` returned the token `{` and a function f_2. The function f_2 when applied to `()` returned the token `{` and a function f_3. The function f_3 when applied to `()` returned the token `1` and a function f_4. And so on. This was clumsy and inefficient. Furthermore, its enforced linearity as a list of tokens was completely at odd with the inherent tree-like nested nature of

Kleisli's data model. Thus many "lazy" operators of CPL in the old 1994 prototype of Kleisli had complicated implementation and high overhead.

The current release of Kleisli has a much better implementation of laziness that is simpler, more efficient, more flexible, and more natural. In this new implementation, a set is represented in a tree-like fashion, where branches that are lazy and branches that are not lazy can be freely mixed. This representation does not have the inconveniently imposed linearity of the tokens of the 1994 old prototype. Thus many operators in the new Kleisli have implementations that work uniformly over objects that are lazy and objects that are not lazy.

6.2 Concurrency

The 1994 old prototype was strictly sequential. It did not exploit network latency inherent in access to remote data sources. It did not exploit parallelism made possible by using several remote data sources, which usually resided on distinct independent servers. It did not exploit parallelism even when it was run on a large machine with multiple processors.

The current release of Kleisli is inherently concurrent. It is able to automatically overlap the requests it dispatches to a remote source, as soon as the source is free, without needing to wait until it has completely received and processed the previous reply from the remote source. (Note that replies from the kind of remote sources accessed by Kleisli are usually sets containing many elements that can be processed in a data-parallel manner.) It aggressively exploits the parallelism made possible by sending requests to multiple remote data sources simultaneously and by scheduling tasks to run on multiple processors. Recall from Section 5 that a simple two-fold concurrency on `sortez` produced a near two-fold increase in query performance. Kleisli and many of its sources are typically run on Enterprize-level servers with large memory and many processors. These machines can easily sustain an order of magnitude more concurrency. As many queries handled by Kleisli are inherently data parallel, this increase in concurrency typically translates to an order of magnitude increase in query performance.

6.3 Types

The type system of the 1994 old prototype was poorly implemented. Its treatment of record types was the main source of inefficiency. Record types were implemented based on Remy's representation (Remy, 1992; Wong, 1995a). If this was properly exploited, it would be possible to test within constant time whether two record types had the same fields; and it would be possible to extract within constant time the type of any given field of a record type. However, in the 1994 old prototype the author failed to properly exploit this representation. In fact, whenever a record type was used, it was first (unnecessarily) converted from Remy's representation into a list of field name-field type pairs; and whenever a record type was stored, it was always (unnecessarily) recalculated from the list representation into the Remy representation at great cost. The current release of Kleisli has a completely over-

hauled type system and has fully rectified this mistake. The difference in the type inference performance of the two implementations is very significant. In particular, those examples that Kosky (1996) reported to take hours to infer their types using the old prototype, took only minutes using the current implementation on a much less powerful machine.

Besides improvement in type inference performance, the type system of the current release of Kleisli also has more functionalities than the old 1994 prototype. The old prototype did not allow the introduction of user-defined abstract data type. It did not support subtyping. It did not support "tuple" type variables. The current release of Kleisli has all of these features. "Tuple" type variables have been very useful for implementing functions on arrays. However, abstract data type and subtyping has not been used in any of our bioinformatics applications.

The old 1994 prototype also did not make use of meta information of the input-output type of a server when such information was available to achieve better type safety. The current release of Kleisli makes use of such information as much as possible. Note that the output type of a server is usually dependent on the input request. For example, the output type of a relational database server depends on the input SQL query. As each relational database server can in principle produce an infinite number of different types, it is in general not possible to have one separate strongly-typed interface function for each possible input-output type of a server. Therefore, it is often more practical to sacrifice some type safety in exchange for a small number of interface functions for each server. A type system that is able to type check a function based on its actual supplied input at compile time can recover some of the type safety lost in this compromise. The type system of the current release of Kleisli has this capability.

Example 6.1
Consider the following CPL program, where `gdb` is a connection to the relational database GDB (Pearson *et al.*, 1992) containing human genome mapping information from Section 5.

```
primitive locus-sym ==
  process "select id = x.#locus_id, sym = x.#locus_symbol
           from locus x
           where 1 = 1"
  using gdb;
```

The old 1994 prototype would accept it as typeable and produces `{''1}`, meaning a polymorphic set whose element type can be instantiated to any type. This was not type safe in two ways. First, obviously, `locus-sym` could only be safely used as a set of records, each having a `#id` field and a `#sym` field. Second, it could only run if the `locus` table in GDB indeed had these two fields.

In constrast, the current release of Kleisli would interrogate the catalogue information from GDB. If `locus` indeed had the two required fields, then it would produce the type `{(#id: ''4, #sym: ''3)}`, which would ensure that `locus-sym` could only be used as a set of records, each having a `#id` field and a `#sym` field. This checking is possible because all relational database system keeps a "catalogue" that

provides various meta information on its tables, such as the names of columns, the availability of indices on columns, expected sizes, etc. □

In fact, the current release of Kleisli can figure out the proper output type of a server even when the input value is not fully supplied at compile time. The necessary meta information needed to achieve this capability, if available, is recorded by the Driver Manager Module depicted in Subsection 2.1 when a driver is registered into Kleisli.

Example 6.2
Let us use the interface to scop from Subsection 3.3 for illustration. Consider the following CPL program.

```
primitive S1 == \x1 => process x1 using scop;
primitive S2 == \x2 => process <#sidinfo: x2> using scop;
primitive S3 == \x3 => process <#numsid: x3> using scop;
```

The type produced for `S1` is `<#sidinfo: string, #sidallinfo: unit, #lssid: unit, #numsid: string, #pdbsid: string> -> {''3}`. Thus `x1` is one of four possible variants, representing the four possible kinds of requests supported by scop. The output type of `S1` is a polymorphic set, as without knowing the actual request type of `x1`, it is not possible to provide more information.

The type produced for `S2` is `string -> {(#sid: string, #pdb: string, #num: string, #region: string, #type: (#cl: string, #cf: string, #sf: string, #fa: string, #sp:string), #desc: (#cl: string, #cf: string, #sf:string, #fa: string, #dm: string, #sp: string))}`. The context `<#sidinfo: x2>` is sufficient for the current release of Kleisli to determine that `S2` is a function that sends a request for a scop entry corresponding to the input *sid* `x2`. It is unnecessary to know the actual value of `x2`. The output type of `S2` is therefore the type of a scop entry, which is available as a piece of meta information to Kleisli.

Similarly, the type produced for `S3` is `string -> {string}`. The context `<#numsid: x3>` is sufficient for the current release of Kleisli to determine that `S3` is a function that sends a request for all *sid*'s of scop entries belonging the the input scop classification `x3`. The output type of `S3` is thus that of a set of *sid*'s, which is available to Kleisli as a piece of meta information and happens to be `{string}`. □

6.4 Optimization

The optimizer of the 1994 old prototype was simple minded. It had only one phase, one rule base, and one rule application strategy that kept rewriting a query in a top-down manner using rules in the rule base until a normal form was reached. It also had no support routines for analysis and manipulation of abstract syntax objects. In contrast, the current release of Kleisli has a more matured optimizer. As mentioned in Subsection 4.3, it has an extensible number of phases, each with its own rule base and rule application strategy. In addition, it has adequate support for analysis and manipulation of abstract syntax objects, such as (conservative) testing of equivalences and so on. The simple-minded rigidity of the old optimizer

was responsible for several further shortcomings, to be described shortly, which are overcome by the flexibility of the current optimizer.

As said earlier, the old prototype had only one rule application strategy. Furthermore, it was not implemented in such a way new rule application strategies could be easily produced. In contrast, as shown in Subsection 4.3, the new optimizer carefully divided the implementation of strategies into a common traversal component and an individually-specialized control component. As a result, it is relatively simple to generate new strategies in the new optimizer.

Many important optimization rules were also missing from the old prototype. For example, it did not have rules for join re-ordering and for migrating joins across two distinct relational databases. Both of these rules and several others are present in the new optimizer. The new optimizer thus is able to produce significantly better optimized queries that involve more than two relational databases. Incidentally, the example in Section 5 also involved a join across two different relational databases and thus could not be optimized by the old prototype.

The old prototype needed many rules to accomplish some optimizations due to its rigidity of having only one rule application strategy and one rule base. For example, it needed more than thirty rules to specify how to migrate selections, projections, and joins to a single external relational database system, as shown in the author's 1994 thesis (Wong, 1994). In contrast, the new optimizer has only three rules to accomplish these three optimizations, as shown in Subsection 5.5 of (Wong, 1995c). Moreover, exploiting the `ContextSensitive` rule generator shown in Example 4.6, each of these three rules in the new optimizer is not much more complicated than any of the corresponding thirty-plus rules in the old prototype! Let us use the rule for migrating projections for illustration. A special case of this rule is to rewrite `{ x.#name | \x <- process "select * from T x where 1 = 1" using A }` to `{ x.#name | \x <- process "select name = x.name from T x where 1 = 1" using A }`, assuming `A` connects to a SQL database. 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 6.3
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 4.6.

```
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
```

Recall from Subsection 4.2 that `process` M `using` S is represented in the NRC Module as a `SYN` object `Read`(S, p, M), where p is a priority to be assigned by Kleisli. 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
```

This implementation of `MigrateProj` requires M to be a string. The actual implementation in the new optimizer allows M to be any `SYN` object; see (Wong, 1995c). □

6.5 Drivers

The 1994 old prototype had only two drivers. The first was a driver for the ASN.1 version of Entrez. The second was a driver for Sybase database systems. Both drivers were developed by Kyle Hart. These two drivers provided the old prototype access to a very small number of external sources, greatly limiting the number of bioinformatics problems that it could be directly applied to.

The current release of Kleisli provide drivers for more than sixty different kinds of bioinformatics sources, covering a significant proportion of the currently publicly available bioinformatics sources. Such a significant coverage of bioinformatics tools and databases makes the current version of Kleisli directly applicable to a much larger number of bioinformatics integration problems.

6.6 Interfaces

The 1994 old prototype came with only one interface: the high-level query language CPL. This is adequate if one is only interested in database-style queries. However, it is important that additional interfaces to Kleisli become available.

A group of our users are traditional database programmers who prefer the more familiar syntax of SQL to the more "mathematical" syntax of CPL. As it is well known (Date, 1984) that SQL is a poorly designed query language despite its popularity, the author resisted the demand of these users for the last three years. However, the author recently gave in partially. The current release of Kleisli provides a second high-level query language sSQL (short for simplified SQL), whose syntax is a sanitized version of SQL. sSQL is described in full in (Wong, 1998b).

It is implemented in Kleisli by translating to NRC, just like CPL. sSQL uses SQL-like constructs such as `select-from-where`, `select-distinct-from-where`, `select-from-where-order by`, `select-distinct-from-where-order by`, etc. instead of the comprehension of CPL. However, sSQL does not use SQL-like syntax for grouping and aggregate functions. These special grouping and aggregate function constructs are needed in SQL because SQL's data model does not allow nesting, which is not the case in sSQL.

Another group of our users are non-programmers who prefer to specify queries through a graphic interface. The current release of Kleisli has an experimental graphic interface called QUICK (Tan *et al.*, 1998) that allows queries to be specified by pointing and clicking on a graphic representation of data sources. A more robust and powerful version of QUICK has already been designed and will be released at end of 1999.

Many of our users also want to reformat the output produced by Kleisli for viewing with a web browser. The current release of Kleisli comes with several additional interfaces that integrate its data model, exchange format, and other facilities to the two most popular programming languages for the web: Java and Perl. A description of these interfaces is given in (Chen & Wong, 1998). These interfaces allow a Java or Perl programmer to directly access Kleisli, CPL, or sSQL, and manipulate the results as native complex objects of Java or Perl, as opposed to as raw strings. He can thus easily reformat the output produced by Kleisli for viewing with his web browser or for other purposes. Similar interfaces to other programming languages have also been planned.

7 Conclusion

The Kleisli system and its high-level query language CPL embody many advances made in database query languages and in functional programming. It represents a significant deployment of functional programming in an industrial strength prototype that has made significant impact on data integration in bioinformatics (Benton, 1996; Karp, 1996; Baker & Brass, 1998). Indeed, since the early Kleisli prototype was applied to bioinformatics(Hart *et al.*, 1994b), it has been used to efficiently solve many real-life data integration problems in bioinformatics (Baker *et al.*, 1998; Chen *et al.*, 1998a; Chen *et al.*, 1998b; Chen *et al.*, 1998c). To date, thanks to the use of CPL, we do not know of another system that can express general bioinformatics queries as succintly as Kleisli.

There are several key ideas behind the success of the system. The first is its use of 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 CPL which allows these objects to be manipulated easily. The third is its use 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. The influence of functional programming research on these ideas was already described.

There is one last reason behind the success of the system. In spite of the so-

phistication of the Kleisli system, it has a remarkably compact implementation, consisting of about 45000 of codes in Standard ML of New Jersey. This compares well to the 1000000 lines of C codes for a typical full-blown commercial database system such as Oracle, even after taking into consideration that a large proportion of these 1000000 lines are devoted to transaction control, disk management, and user interfaces. The implementor (this author) has no doubt that without this robust platform of functional programming, it would have demanded much more effort in implementing Kleisli.

Acknowledgements. The author designed and implemented the first prototype of Kleisli/CPL in 1994, while he was at the University of Pennsylvania. Peter Buneman, Val Tannen, Leonid Libkin, and Dan Suciu, collaborated with him on query language theory and on foundational issues of CPL. Susan Davidson and Chris Overton introduced the author to problems in bioinformatics. Kyle Hart helped him in applying Kleisli to address his first bioinformatic integration problem. He 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, Jing Chen, and Jiren Wang added much to its bioinformatics support under funding from the Singapore Economic Development Board. The author has begun seriously using Kleisli in bioinformatics himself towards end of 1997. Nam-Hai Chua of the Rockefeller University, Anthony Ting of the Institute of Molecular and Cell Biology, and Daphna Strauss of Kent Ridge Digital Labs (now at the Israel Embassy in Moscow) taught him almost everything he knows on the biology relevant to this exciting subject. The author is grateful to all of them for their contributions. He would also like to thank Phil Wadler for bringing Mike Spivey's work to his attention, for encouraging him to write this article, and for suggestions that much improved the presentation of this article.

References

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison-Wesley.

Altschul, S. F., & Gish, W. (1996). Local alignment statistics. *Methods in Enzymology*, **266**, 460–480.

Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, **25**(17), 3389–3402.

Baker, P. G., & Brass, A. (1998). Recent development in biological sequence databases. *Current Opinion in Biotechnology*, **9**, 54–58.

Baker, P. G., Brass, A., Bechhofer, S., Goble, C., Paton, N., & Stevens, R. 1998 (June). TAMBIS—transparent access to multiple bioinformatics information sources. *Pages 25–34 of: Proceedings of 6th International Conference on Intelligent Systems for Molecular Biology.*

Benton, D. (1996). Bioinformatics—principles and potential of a new multidisciplinary tool. *Trends in Biotechnology*, **14**(August), 261–272.

Breazu-Tannen, V., & Subrahmanyam, R. (1991). Logical and computational aspects of programming with Sets/Bags/Lists. *Pages 60–75 of: LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming*. Springer Verlag.

Breazu-Tannen, V., Buneman, P., & Naqvi, S. (1991). Structural recursion as a query language. *Pages 9–19 of: Proceedings of 3rd International Workshop on Database Programming Languages*. Morgan Kaufmann.

Buneman, P., Naqvi, S., Tannen, V., & Wong, L. (1995). Principles of programming with complex objects and collection types. *Theoretical Computer Science*, **149**(1), 3–48.

Burks, C., Cinkosky, M.J., Fischer, W.M., Gilna, P., Hayden, J.E., Keen, G.M., Kelly, M., Kristofferson, D., & Lawrence, J.D. (1992). GenBank. *Nucleic Acids Research*, **20 Supplement**, 2065–9.

Cattell, R. G. G. (ed). (1996). *The Object Database Standard: ODMG-93*. San Mateo, California: Morgan Kaufmann.

Chen, J., & Wong, L.. (1998). *The Kleisli/Perl reference manual: A data exchange format and some supporting tools*. Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613.

Chen, J., Chua, N.-H., Strauss, D., & Wong, L. (1998a). Extracting Kozak consensus sequence using Kleisli. *Pages 218–223 of: Proceedings of 1st International Conference on Bioinformatics of Genome Regulation and Structure*.

Chen, J., Zhang, L., & Wong, L.. (1998b). A protein patent query system powered by Kleisli. *Pages 593–595 of: Proceedings of ACM SIGMOD International Conference on Management of Data*.

Chen, J., Strauss, D., & Wong, L. (1998c). Using Kleisli to bring out features in BLASTP results. *Pages 102–111 of: Genome Informatics 1998*. Tokyo, Japan: Universal Academy Press.

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

Date, C. J. (1984). A critique of the SQL database language. *SIGMOD Record*, **14**(3), 8–52.

Davidson, S., Overton, C., Tannen, V., & Wong, L. (1997). BioKleisli: A digital library for biomedical researchers. *International Journal of Digital Libraries*, **1**(1), 36–53.

Dong, G., Libkin, L., & Wong, L. (1997). Local properties of query languages. *Pages 140–154 of: Proceedings of 6th International Conference on Database Theory*.

Frakes, W. B., & Baeza-Yates, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Prentice Hall.

Goldberg, A., & Paige, R. (1984). Stream processing. *Pages 53–62 of: Proceedings of ACM Symposium on LISP and Functional Programming*.

Hart, K. W., Searls, D. B., & Overton, G. C. (1994a). SORTEZ: A relational translator for NCBI's ASN.1 database. *Computer Applications in the Biosciences*, **10**(4), 369–378.

Hart, K., Wong, L., Overton, C., & Buneman, P. (1994b). Using a query language to integrate biological data. *Abstracts of 1st Meeting on the Interconnection of Molecular Biology Databases*.

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

Karp, P. D. (1996). Database links are a foundation for interoperability. *Trends in Biotechnology*, **14**, 273–279.

Kosky, A. (1996). *Transformaing Databases with Recursive Data Structures.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Leser, U., Lehrach, H., & Crollius, H. R. (1998). Issues in developing integrated genomic databases and application to the human X chromosome. *Bioinformatics*, **14**(7), 583–590.

Libkin, L., & Wong, L. (1997). Query languages for bags and aggregate functions. *Journal of Computer and System sciences*, **55**(2), 241–272.

Litwin, W., Mark, L., & Roussopoulos, N. (1990). Interoperability of multiple autonomous databases. *ACM Computing Surveys*, **22**(3), 267–293.

Lodish, H., Baltimore, D., Berk, A., Zipursky, S. L., Matsudaira, P., & Darnell, J.. (1995). *Molecular Cell Biology.* New York: W. H. Freeman.

Murzin, A., Brenner, S. E., Hubbard, T., & Chothia, C. (1995). SCOP: A structural classification of protein database for the investigation of sequences and structures. *Journal of Molecular Biology*, **247**, 536–540.

National Center for Biotechnology Information. (1992). *NCBI ASN.1 Specification.* Revision 2.0.

Ngu, A., Yan, L., & Wong, L. (1993). Heterogeneous query optimisation using maximal subqueries. *Pages 413–420 of: Proceedings of 3rd International Symposium on Database Systems for Advanced Applications.*

Ohori, A., Buneman, P., & Breazu-Tannen, V. (1989). Database programming in Machiavelli, a polymorphic language with static type inference. *Pages 46–57 of: Proceedings of ACM-SIGMOD International Conference on Management of Data.*

Papakonstantinou, Y., Garcia-Molina, H., & Widom, J. (1995). Object exchange across heterogenous information sources. *Pages 251–260 of: Proceedings of IEEE International Conference on Data Engineering.*

Paulson, L. C. (1983). A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–49.

Pearson, P., Matheson, N., Flescher, N, & Robbins, R. J. (1992). The GDB human genome data base anno 1992. *Nucleic Acids Research*, **20**, 2201–2206.

Pearson, W. R. (1995). Comparison of methods for searching protein sequence databases. *Protein Science*, **4**, 1145–1160.

Remy, D. (1989). Typechecking records and variants in a natural extension of ML. *Pages 77–88 of: Proceedings of 16th Symposium on Principles of Programming Languages.*

Remy, D. (1992). Efficient representation of extensible records. *Pages 12–16 of:* Lee, P. (ed), *Proceedings of ACM SIGPLAN Workshop on ML and its Applications.*

Schuler, G. D., Epstein, J. A., Ohkawa, H., & Kans, J. A. (1996). Entrez: Molecular biology database and retrieval system. *Methods in Enzymology*, **266**, 141–162.

Selletin, J., & Mitschang, B. (1998). Data-intensive intra- & internet applications—Experiences using Java and CORBA in the World Wide Web. *Pages 302–311 of: Proceedings of 14th IEEE International Conference on Data Engineering.*

Siegel, J. (1997). *CORBA: Fundamentals and Programming.* New York: Wiley.

Spivey, M. (1990). A functional theory of exceptions. *Science of Computer Programming*, **14**, 25–42.

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

Suciu, D., & Wong, L. (1995). On two forms of structural recursion. *Pages 111–124 of: LNCS 893: Proceedings of 5th International Conference on Database Theory.* Prague: Springer-Verlag.

Tan, W. C., Wang, K., & Wong, L. (1998). A graphical interface to genome multidatabases. *Journal of Database Management*, **9**(1), 24–32.

Ullman, J. D. (1989). *Principles of Database and Knowledgebase Systems II: The New Technologies*. Rockville, MD 20850: Computer Science Press.

Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science*, **2**, 461–493.

Wadler, P. (1998). Functional programming: An angry half-dozen. *SIGPLAN Notices*, **33**(2), 25–30.

Walsh, S., Anderson, M., & Cartinhour, S.W. (1998). ACEDB: A database for genome information. *Methods Biochem Anal*, **39**, 299–318.

Wong, L. (1994). *Querying Nested Collections*. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Wong, L. (1995a). An introduction to Remy's fast polymorphic projection. *ACM SIGMOD Record*, **24**(3), 34–39.

Wong, L. (1995b). *The Kleisli Query System Reference Manual*. Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613.

Wong, L. (1995c). *The Kleisli/CPL Extensible Query Optimizer Programmer Guide*. Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613.

Wong, L. (1998a). *The Collection Programming Language Reference Manual*. Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613.

Wong, L. (1998b). *The Simplified SQL Reference Manual*. Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613.