The Functional Guts of the Kleisli 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. 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. This paper describes the influence of functional programming research that benefits the Kleisli system, especially the less obvious ones at the implementation level.

1 Introduction

The Kleisli system [14] is an advanced broad-scale integration technology that has proved useful in the bioinformatics arena. 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, 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.

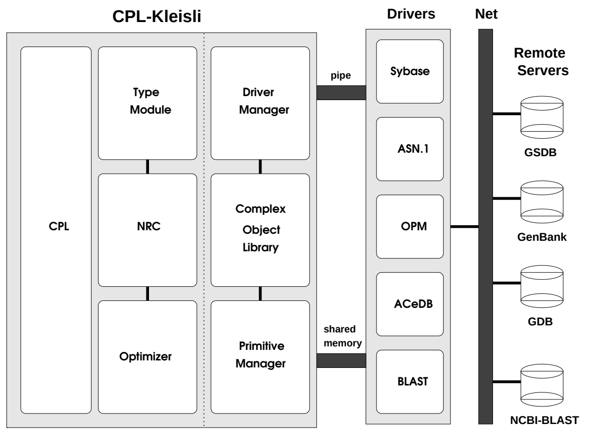
This paper provides an overview of the Kleisli system; a summary of its impact on query language theory; and a description of the influence of functional programming research that benefits the Kleisli system, especially the less obvious ones at the implementation level. The organization of the paper 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 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. Section 4 discusses the impact of Kleisli on bioinformatics data integration. In particular, the first Kleisli query written for this purpose is reproduced here to illustrate the smoothness of Kleisli's interface to relational and non-relational bioinformatics sources and its optimizations.

This paper is largely extracted from a much longer and detailed paper [36] in *J. Funct. Prog.* Additional topics discussed in the longer paper include laziness, concurrency, "dependency-like" typing, and optimizations for relational databases.

2 Quick Tour of Kleisli

We begin with the data model of Kleisli that is based on complex object types and with the high-level query language supported by Kleisli called CPL, which stands for Collection Programming Language. Its 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[14] 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 version that forms the backbone of the ConnectivityEngine^{TM} of Kris Technology Inc. (www.kris-inc.com) contains over sixty drivers for all popular bioinformatics systems, including Sybase, Oracle, Entrez [27], WU-BLAST2 [1], Gapped BLAST [2], ACEDB [33], etc. Also, the optimizer of Kleisli can be customized by 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 [8], and is chosen as the internal query representation because it is easy to manipulate and amenable to machine analysis. The NRC expression is then analyzed 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.

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 [13]. It allows arbitrarily nested records, sets, lists, bags, and variants. A variant is also called a tagged union type and represents a type that is “either this or that”. 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 real life applications. Firstly, the particular bulk data type 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 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 [31] 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. While an amino acid sequence is a string of letters, certain regions and positions of the string are of special* *biological interest, such as binding sites, domains, and so on. The feature table of a GenPept report is the part of the GenPept report that documents the positions of these regions of special biological interest, as well as annotations or comments on these regions. The following type represents the feature table of a GenPept report from Entrez [27].*

```
(#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 *Here is the feature table of GenPept report 131470, a tyrosine phosphatase 1C sequence.*

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

The particular feature displayed above 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 is a cross reference to the US National Center for Biotechnology Information taxonomy database. □

The schemas and structures of all popular bioinformatics databases, flat files, and softwares are easily mapped into this data model. At the extreme of data structure complexity are Entrez [27] and ACEDB [33], 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 [13] such as Sybase and Oracle, which contain flat sets of records. Currently, Kleisli gives access to over sixty of these and other bioinformatics sources. The reason for this ease of mapping bioinformatics sources to Kleisli's data model is that they are all inherently composed of combinations of sets, bags, lists, records, and variants. 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.

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.

In short, 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 [25], 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 [10]. An interesting feature of the syntax of CPL is the heavy use of the comprehension syntax, which showed up long ago in functional programming languages and later formalized by Wadler [32]. 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, we illustrate CPL by a few examples on a set of feature tables `DB`.

Example 2.3 *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%" };
```

□

This query is a simple project-select query. A project-select query is a query that operates on one (flat) relation or set. Thus the transformation that such a query can perform is limited to selecting some elements of the relation and extracting or projecting some fields from these elements. Except for the fact that the source data and the result may not be in first normal form, these queries can be expressed in a relational query language. However, 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 2.4 *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 2.5 *This query demonstrates how to do nesting in CPL. The subquery* `DB'` *is the restructuring of* `DB` *by pairing each entry with its source organism. 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 [6] 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 [7]. 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 [32]. A nested relational calculus, which is denoted here by $\mathcal{NRC}$, was then designed around this restriction [8]. $\mathcal{NRC}$ is essentially the simply-typed lambda calculus extended by a construct for building records, a construct for decomposing records by field selection, a construct for building sets, a construct for decomposing sets by means of the restriction on structural recursion. Specifically, the construct for decomposing sets is $\bigcup\{e_1 \mid x \in e_2\}$, which forms a set by taking the big union of $e_1[o/x]$ over each o in the set e_2. $\mathcal{NRC}$ (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[32].

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 [29, 15, 18, 8, 30]. 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 [8]. $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ was shown to capture the power of SQL, including aggregate functions and group-by constructions [18]. 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[18] 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.

3 Influence of Functional Programming

Functional programming has a significant influence on 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 [21, 25]. Although CPL's view of record type variables is closer to that in Machiavelli [21], than the row variables of Remy [25], the implementation [35] of polymorphic records in Kleisli is based on a clever idea of Remy [26]. 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 [8] that the core of CPL is founded on structural recursion [6] and monad [32].

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 3.1, the abstract internal representation of queries in Subsection 3.2, and the optimizer in Subsection 3.3.

3.1 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.

Three of these requirements are addressed by 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 is applicable 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 [17] data from Entrez, 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 any type to be declared at all. 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 [20]—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 [17] 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 [22, 19]. However, Kleisli's is probably the first self-describing exchange format that is consciously derived from type inference!

3.2 Kleisli Triples and Abstract Syntax

Let us briefly recall the restricted form of structural recursion which corresponds to the presentation of monads by Kleisli [32, 8]. 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 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 [16], 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

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

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.

Let us now consider the `ext` construct. As pointed out by Wadler [32], 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

$$\texttt{ext}\{e_1 \mid \backslash x \texttt{ <- } \texttt{ext}\{e_2 \mid \backslash y \texttt{ <- } e_3\}\} \rightsquigarrow$$
$$\texttt{ext}\{\ \texttt{ext}\{e_1 \mid \backslash x \texttt{ <- } e_2\} \mid \backslash y \texttt{ <- } e_3\}$$

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. 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:

```
datatype SYN = ...
| EmptySet
| SngSet of SYN
| UnionSet of SYN * SYN
| ExtSet of SYN * VAR * SYN
| IfThenElse of SYN * SYN * SYN
```

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

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

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

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

Example 3.1 *Vertical loop fusion.*

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

□

The Kleisli optimizer also performs many other optimizations. These optimizations include a 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. See [36] for more details.

3.3 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 [30]. The optimizer 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. 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 3.2 *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 Qry =
  case Pick RDB Qry
  of SOME ImprovedQry => ImprovedQry
  | NONE => Decompose (MaxOnce RDB) Qry
```

□

Example 3.3 *The* `BottomUpOnce` *strategy applies rules in a leaves-to-root pass. It tries to rewrite each node at most once as it moves towards the root of the query expression. Here is its "control" component:*

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

□

Let us now present an interesting class of rules that requires the use of multiple rule application strategies. The scope of rules like the vertical loop fusion in the previous 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 3.4 *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 nontermination otherwise. The second clause says that the rule should be applied to all other forms of* `IfThenElse`*. The third clause says that the rule is not applicable in any other situation.*

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

The second clause is the meat of the implementation. The inner part of the rewrite `if` e_1 `then` ... e_1 ...

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

```
fun ContextSensitive Rule Strategy Qry =
let (* This flag is set if Rule is applied *)
  val Changed = ref false
  (* Set up a context-sensitive rule base *)
  val RDB = {
    DoTrace = ref true,
    Trace = ref (fn _ => fn _ => fn _ =>
      (* Changed is true if Rule is used *)
      Changed := true)
    Rules = ref [("", Rule)]}
  (* Apply Rule using Strategy. *)
  val OptimizedQry = Strategy RDB Qry
in if !Changed then SOME OptimizedQry 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, compared to the original optimizer from [34]. The author is *not* the first to discover this particular method of implementing rewrite strategies; Paulson [23] and Spivey [28] presented similar ideas.

4 Impact on Bioinformatics

"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" [3].

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.

It is now widely agreed that Kleisli has significantly reduced the difficulty of integrating biology data [5, 3]. To get a sense of Kleisli's impact on bioinformatics, let us describe the very first bioinformatics query implemented in Kleisli in 1994 [14]. It was one of the so-called "impossible" queries of a US Department of Energy Bioinformatics Summit Report (`www.gdb.org/Dan/DOE/whitepaper/contents.html`.) 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, the query means that for each gene in a particular position in the human genome, find DNA sequences from non-human organisms that are similar to it.

In 1994, the main database containing cytogenetic band information was the GDB [24], which was a Sybase relational database. In order to find homologs, the actual DNA sequences were needed and the ability to compare them was also needed. Unfortunately, that database did not keep actual DNA sequences. The actual DNA sequences were kept in another database called GenBank [9]. At the time, access to GenBank was provided through the ASN.1 version of Entrez [27], which was an extremely complicated retrieval system. Entrez also kept precomputed homologs of GenBank sequences.

So this query needed the integration of GDB (a relational database located in Baltimore) and Entrez (a non-relational "database" located in Bethesda) that first extracted names of genes on the desired cytogenetic band and accessed Entrez for homologs of these genes and finally filtered these homologs to retain the non-human ones. This query was considered "impossible" as there was at that time no system that could work across the bioinformatics sources involved due to their heterogeneity, complexity, and geographical locations. Given the complexity of this query, the CPL query given in [14] was remarkably short. Since then Kleisli has been used to power many bioinformatics applications [4, 12, 11, etc.]

Example 4.1 *The query mentioned is shown below.*

```
sybase-add (#name:"gdb", ...);
readfile locus from "locus_cyto_location" using gdb;
readfile eref from "object_genbank_eref" using gdb;
{(#accn: g.#genbank_ref, #nonhuman-homologs: H)
| \c <- locus, c.#chrom_num = "22",
  \g <- eref, g.#object_id = c.#locus_id,
  \H == { u
   | \u <- na-get-homolog-summary(g.#genbank_ref),
     not(u.#title string-islike "%Human%"),
     not(u.#title string-islike "%H.sapien%")},
  not (H = { })}
```

The first three lines connected to GDB and mapped two tables in GDB to Kleisli. After that, these two tables could be referenced within Kleisli as if they were two locally defined sets, `locus` *and* `eref`*. The next 9 lines extracted from these tables the accession numbers of those genes on Chromosome 22 and used the Entrez function* `na-get-homolog-summary` *to obtain their homologs and filtered these homologs for non-human ones.*

Besides the obvious smoothness of integration of the two data sources, this query was also remarkably efficient. On the surface, it seemed to fetch the `locus` *table in its entirety once and the* `eref` *table in its entirety n times from GDB, as a naive evaluation of the comprehension would be two nested loops iterating over these two tables. Fortunately, in reality, the Kleisli optimizer was able to migrate the join and projections on these two tables into a single efficient access to GDB.* □

5 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. Indeed, since the early Kleisli prototype was applied to bioinformatics, it has been used to efficiently solve many real-life data integration problems in bioinformatics. 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.

Those who have read [14] would notice that the SQL flavor in the original implementation [14] has completely vanished. This is because the current version of Kleisli has made significant advancement in interfacing relational databases.

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 sophistication 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. Peter Buneman, Val Tannen, and Leonid Libkin—for many fruitful collaboration.

References

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

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

[3] P. G. Baker and A. Brass. Recent development in biological sequence databases. *Curr. Op. Biotech.*, 9:54–58, 1998.

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

[5] D. Benton. Bioinformatics — principles and potential of a new multidisciplinary tool. *TIBTECH*, 14:261–272, 1996.

[6] V. Breazu-Tannen et. al. Structural recursion as a query language. *DBPL*, 3:9–19, 1991.

[7] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. *ICALP*, 18:60–75, 1991.

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

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

[10] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1996.

[11] J. Chen et. al. Using Kleisli to bring out features in BLASTP results. *Genome Informatics*, 9:102–111, 1998.

[12] J. Chen et. al. A protein patent query system powered by Kleisli. *ACM SIGMOD Record*, 27(2):593–595, 1998.

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

[14] S. Davidson et. al. BioKleisli: A digital library for biomedical researchers. *Int. J. Digital Libraries*, 1(1):36–53, 1997.

[15] G. Dong et. al. Local properties of query languages. *ICDT*, 6:140–154, 1997.

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

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

[18] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *JCSS*, 55(2):241–272, 1997.

[19] W. Litwin et. al. Interoperability of multiple autonomous databases. *ACM Comput. Surveys*, 22(3):267–293, 1990.

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

[21] A. Ohori et. al. Database programming in Machiavelli, a polymorphic language with static type inference. *ACM SIGMOD Record*, 18(2):46–57, 1989.

[22] Y. Papakonstantinou et. al. Object exchange across heterogenous information sources. *ICDE*, 11:251–260, 1995.

[23] L. C. Paulson. A higher-order implementation of rewriting. *Sci. Comput. Prog.*, 3:119–49, 1983.

[24] P. Pearson et. al. The GDB human genome data base anno 1992. *NAR*, 20:2201–2206, 1992.

[25] D. Remy. Typechecking records and variants in a natural extension of ML. *POPL*, 16:77–88, 1989.

[26] D. Remy. Efficient representation of extensible records. In *Proc. of ACM SIGPLAN Workshop on ML and its Applications*, pages 12–16, 1992.

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

[28] M. Spivey. A functional theory of exceptions. *Sci. Comput. Prog.*, 14:25–42, 1990.

[29] D. Suciu. Bounded fixpoints for complex objects. *TCS*, 176(1–2):283–328, 1997.

[30] D. Suciu and L. Wong. On two forms of structural recursion. *ICDT*, 5:111–124, 1995.

[31] J. D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies.* Computer Science Press, 1989.

[32] P. Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2:461–493, 1992.

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

[34] L. Wong. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994.

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

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