The Kleisli Approach to Data Transformation and Integration

Susan B. Davidson
University of Pennsylvania
susan@central.cis.upenn.edu

Limsoon Wong
Kent Ridge Digital Labs
limsoon@krdl.org.sg

February 7, 2001

Abstract

Kleisli is a data transformation and integration system that can be used for any application where the data is typed, but has proven especially useful for bioinformatics applications. It extends the conventional flat relational data model supported by the query language SQL to a complex object data model supported by the collection programming language CPL. It also opens up the closed nature of commercial relational data management systems to an easily extensible system that performs complex transformations on autonomous data sources that are heterogeneous and geographically dispersed. This paper describes some implementation details and example applications of Kleisli.

1 Introduction

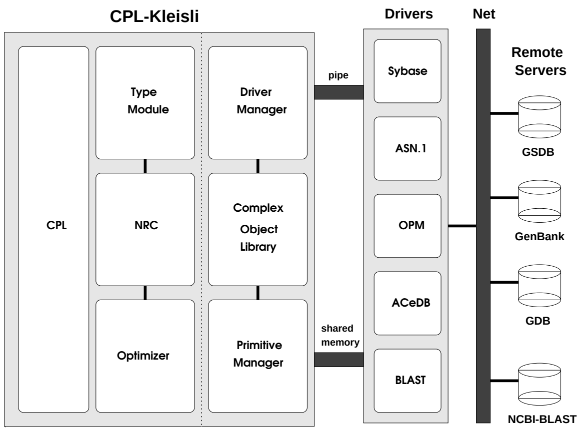
The Kleisli system [14, 32, 33] is an advanced broad-scale integration technology that has proven very useful in the bioinformatics arena. Many bioinformatics problems require access to data sources that are large, highly heterogeneous and complex, constantly evolving, and geographically dispersed. Solutions to these problems usually involve many steps and require information to be passed smoothly (and usually transformed) between the steps. Kleisli is designed to handle these requirements directly by providing a high-level query language, CPL, that can be used to express complicated transformations 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). Even the data format that Kleisli uses to exchange information with the external world is derived from ideas in type inference.

This paper provides an overview of the Kleisli system, a summary of its impact on query language theory, a description of its SML-based implementation, a description of its handling of relational databases, as well as some sample applications in the biomedical arena. The organization of the paper is as follows. Section 2 offers an overview of the architecture, data model, and query language (CPL) of Kleisli. Section 3 is a discussion of Kleisli's type system and self-describing data exchange format and their impact on data integration in a dynamic heterogeneous environment. Section 4 describes how monads give rise to Kleisli's internal abstract representation of queries and simple optimization rules. Section 5 explains how higher-order functions give rise to a simple implementation of Kleisli's powerful optimizer. Section 6 gives details on Kleisli's optimization with respect to the most important class of external data sources, viz. relational databases. Section 7 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 to show its optimizations. Section 8 shows how to use Kleisli to turn a flat relational database system into a complex object store to warehouse complex biological data. Section 9 demonstrates the use of Kleisli to access multiple external data sources and external data analysis functions for querying protein patents. Finally, Section 10 uses a clinical database to demonstrate Kleisli's ability to perform "window" queries. Such queries are very clumsy to write in SQL, since the data must first be segmented and then each segment is analyzed separately.

2 Quick Tour of Kleisli

We provide here the complex object data model of Kleisli, and the high-level query language supported by Kleisli called CPL (the Collection Programming Language). Let us begin with the architecture of the system, as depicted in the figure below.



Kleisli is extensible in several ways: It can be used to support 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 of Kleisli that forms the backbone of the ConnectivityEngine^{TM} of GeneticXchange Inc. (www.geneticXchange.com) contains over sixty drivers for many popular bioinformatics systems, including Sybase, Oracle, Entrez [27], WU-BLAST2 [1], Gapped BLAST [3], ACEDB [31], etc. The optimizer of Kleisli can also 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 the abstract calculus NRC. NRC is based on that described in [9], 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 routines. The Complex Object Library contains routines for manipulating complex objects such as code for set intersection and code for set iteration.

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". The collection or "bulk" types – sets, bags, and lists – are homogeneous. In order to mix objects of different types in a set, bag, or list, it is necessary to inject these objects into a variant type.

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. Having only one bulk data type presents at least two problems in real life applications. Firstly, the particular bulk data type may not be a natural model of real data. Secondly, the particular bulk data type may not be an efficient model of real data. For example, if we are restricted to the flat relational data model, the GenPept report in Example 2.1 below must necessarily be split into many separate tables in order to be losslessly stored in a relational database. The resulting multi-table representation of the GenPept report is conceptually unnatural and operationally inefficient. A person querying the resulting data must pay the mental overhead of understanding both the original GenPept report and its badly-fragmented multi-table representation. He may also have to pay the performance overhead of having to re-assemble the original GenPept report from its fragmented multi-table representation to answer queries.

Example 2.1 *The GenPept report is the format chosen by the US National Center for Biotechnology Information to present amino acid sequence information. While an amino acid sequence is a string of letters, certain regions and positions of the string are of special biological interest, such as binding sites, domains, and so on. The feature table of a GenPept report is the part of the GenPept report that documents the positions of these regions of special biological interest, as well as annotations or comments on these regions. The following type represents the feature table of a GenPept report from Entrez [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 one of its fields (`#feature`*) is a set of records, one of whose fields (*`#anno`*) is in turn a list of records. More precisely, it is a record with four fields* `#uid`*,* `#title`*,* `#accession`*, and* `#feature`*. The first three of these store values of types* `num`*,* `string`*, and* `string` *respectively. The* `#uid` *field uniquely identifies the GenPept report. The* `#feature` *field is a set of records, which together form the feature table of the corresponding GenPept report. Each of these records has four fields* `#name`*,* `#start`*,* `#end`*, and* `#anno`*. The first three of these have types* `string`*,* `num`*, and* `num` *respectively. They represent the name, start position, and end position of a particular feature in the feature table. The* `#anno` *field is a list of records. Each of these records has two fields* `#anno_name` *and* `#descr`*, both of type* `string`*. These records together represent all annotations on the corresponding feature.* □

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

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

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 $\texttt{<}l_1 : t_1, ..., l_n : t_n\texttt{>}$ 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; $\texttt{<}l : e\texttt{>}$ for variants; $\{e_1, ..., e_n\}$ for sets; $\{|e_1, ..., e_n|\}$ for bags; and $[e_1, ..., e_n]$ for lists.

Example 2.2 *The feature table of GenPept report 131470, a tyrosine phosphatase 1C sequence, is shown below.*

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

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

The schemas and structures of all popular bioinformatics databases, flat files, and softwares are easily mapped into this data model. At the high end of data structure complexity are Entrez [27] and ACEDB [31], which contain deeply nested mixtures of sets, bags, lists, records, and variants. At the low end of data structure complexity are the relational database systems [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. 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.

We now come to CPL, the primary query language of Kleisli. An interesting feature of the syntax of CPL is the use of the comprehension syntax [8, 30]. An example of 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 *The query below 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 *The following query flattens* `DB` *completely. The sytax* `\a <--- f.#anno` *has similar meaning to* `\x <- DB`*, but works on lists instead of sets. 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, i.e.* `[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 from [6] where structural recursion was presented as a query language. However, structural recursion has two difficulties. The first is that not every syntactically correct structural recursion program is logically well defined [7]. The second is that structural recursion has too much expressive power because it can express queries that require exponential time and space.

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

Thus Tannen and Buneman suggested a natural restriction on structural recursion to reduce its expressive power and to guarantee its well-definedness. Their restriction cuts structural recursion down to homomorphisms on the commutative idempotent monoid of sets, revealing a telling correspondence to monads [30]. A nested relational calculus, which is denoted here by $\mathcal{NRC}$, was then designed around this restriction [9]. $\mathcal{NRC}$ 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[30].

The expressive power of $\mathcal{NRC}$ and its extensions are studied in [28, 15, 19, 9, 29]. 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, while allowing for full-fledged recursion and other operators to be imported easily as needed into the system. $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, \leq^{\mathbb{Q}}, =)$ captures all standard nested relational queries in a high-level manner that is easy for automated optimizer analysis. It is also easy to translate a more user-friendly surface syntax such as the comprehension syntax or the SQL select-from-where syntax into $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, \leq^{\mathbb{Q}}, =)$. It is thus a very suitable core.

3 Type Inference and Self-Describing Exchange Format

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

Three of these requirements are addressed by features of 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, the following value:

```
{ <#name: "John">, <#zip-code: 119613> }
```

This set contains objects of very different structures: a string carrying a `#name` tag and a number carrying a `#zip-code` tag. This feature is particularly useful in handling ASN.1-formatted [18] data from Entrez, one of the most important and most complex sources of DNA sequences, as it contains a profusion of variant types.

In addition, 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 large schemas that take many pages to write down — for example, the ASN.1 schema of Entrez [22]— 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 the previous section. 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. That is, there is no fixed schema and no type declaration. In a sense, each object being exchanged carries its own description. A self-describing format has the important property that, no matter how complex the object being exchanged is, it can be easily parsed and reconstructed without any schema information. To understand this advantage, one should look at the ISO ASN.1 standard [18] open systems interconnection. It is not easy to exchange ASN.1 objects because before we can parse any ASN.1 object, we need to parse the schema that describes its structure first—making it necessary to write two complicated parsers instead of one simple parser.

4 Kleisli Triples and Abstract Syntax

Let us now consider the restricted form of structural recursion which corresponds to the presentation of monads by Kleisli [30, 9]. It is the combinator $ext(\cdot)(\cdot)$ obeying the following three equations:

$$\begin{array}{rcl} ext(f)\{\} & = & \{\} \\ ext(f)\{o\} & = & f(o) \\ ext(f)(A \cup B) & = & ext(f)(A) \cup ext(f)(B) \end{array}$$

Thus, $ext(f)(R)$ is equivalent 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 [17], which corresponds to the physical notion of getting rid of intermediate data and the logical notion of quantifier elimination. Such an optimization on queries in the comprehension syntax can be expressed informally as

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

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 [30], 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

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

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, pointed out to us by Wadler, is that comprehensions require two kinds of terms, expressions and qualifiers, whereas the `ext` formulation requires only one kind of term, expressions.

In order to illustrate this point more concretely, it is necessary to introduce some detail from the implementation of the Kleisli system. Recall from the introductory section that Kleisli is implemented on top of the Standard ML of New Jersey (SML). The type `SYN` of SML objects that represent queries in Kleisli is declared in the NRC Module mentioned in Section 2. The data constructors that are relevant to our discussion are:

```
type VAR = int                     (* Variables, represented by int *)
type SVR = int                     (* Server connections, represented by int *)
type CO = ...                      (* Representation of complex objects *)
datatype SYN = ...
| EmptySet                         (* { }                      *)
| SngSet of SYN                    (* { E }                    *)
| UnionSet of SYN * SYN            (* E1 {+} 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 *)
| Variable VAR                     (* x *)
| Binary (CO * CO -> CO) * SYN * SYN (* Construct for caching
        static objects. This allows the optimizer to insert
        some codes for doing dynamic optimization *)
```

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

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

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

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

Example 4.1 *Vertical loop fusion.*

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

□

5 Higher-Order Functions and Optimization

Another idea that we have exploited in implementing Kleisli is the use of higher-order functions. There are many advantages and conveniences of higher-order functions, besides allowing the expression of better algorithms as discussed in [29]. We use the implementation of the Kleisli query optimizer module for illustration here. 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 ones 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 SML code fragments from the optimizer module mentioned in Section 2 are presented below.

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

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

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

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

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

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

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

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

Example 5.1 *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 5.2 *The* `BottomUpOnce` *strategy applies rules in a leaves-to-root pass. It tries to rewrite each node at most once as it moves towards the root of the query expression. Here is its "control" component:*

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

□

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

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

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

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

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

Example 5.3 *The If-then-else absorption rule is expressed by the* `AborbThen` *rule below. The rule has three clauses. The first clause says that the rule should not be applied to an* `IfThenElse` *whose test predicate is already a Boolean constant, because it would lead to non-termination otherwise. The second clause says that the rule should be applied to all other forms of* `IfThenElse`*. The third clause says that the rule is not applicable in any other situation.*

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

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

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

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

6 Optimization of Queries on Relational Databases

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

Let us begin with the rule for migrating projections. 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 6.1 *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 5.3.*

```
fun FullyProjected x N =
```

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

Recall from Section 4 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
```

□

Second is the rule for migrating selections. A special case of this rule is to rewrite `{x | \x <- process "select * from EMP e where 1 = 1" using A, x.#name = "peter"}` to `{x | \x <- process "select * from EMP e where e.name = 'peter'" using A}`. In the original query, the entire table `EMP` has to be retrieved from the relational database A so that Kleisli can filter for Peter's record. In the rewritten query, the record for Peter is retrieved directly without retrieving any other records. More generally, if `x` is from a relational database and there are some equality tests `x.#`$l_1 = c$, then we should push as many of these tests to the database as possible.

Example 6.2 *The rule for migrating selections to a relational database is implemented by* `MigrateSelect` *below. The rule requires a function* `FlattenTests` *Ok* N *that traverses a tower of if-then-else's in* N *to extract equality tests satisfying the check Ok for migration; it returns a triple* (Pve, Nve, N')*, where* Pve *is a list of tests to be 'and' together,* Nve *is a list of tests to be negated and 'and' together, and* N' *is the remaining (transformed) unflattenable part of the tower; it satisfies* `if` $(\bigwedge Pve) \wedge \neg(\bigvee Nve)$ `then` N' `else` $\{\} = N$*. This function is implemented in a simple way as follows.*

```
fun FlattenTests Ok (IfThenElse (C, N, EmptySet)) =
    let val (Pve, Nve, N') = FlattenTests Ok N
    in if Ok C then (C::Pve, Nve, N') else (Pve, Nve, IfThenElse (C, N', EmptySet)) end
| FlattenTests Ok (IfThenElse (C, EmptySet, M)) =
    let val (Pve, Nve, M') = FlattenTests Ok M
    in if Ok C then (Pve, C::Nve, M') else (Pve, Nve, IfThenElse (C, EmptySet, M')) end
| FlattenTests Ok (ExtSet (N, x, M)) =
    let val (Pve, Nve, N') = FlattenTests Ok N
    in (Pve, Nve, ExtSet (N', x, M)) end
| FlattenTests Ok N = ([], [], N)
```

The `MigrateSelect` *rule is defined below. The function* `SQL.PushTest` *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. The function call* `SQLDict.ExtractLONG` S C *uses the catalog of the relational database* S *to determine* *which of the fields in the output of the SQL query C are BLOB-like; a BLOB-like field is usually a very large string where equality test on it is not supported by the underlying database S. The function* `CanPushTest` *uses the output of* `SQLDict.ExtractLONG` *to produce a function to test if an expression is an equality test that can be migrated.*

```
fun MigrateSelect (ExtSet (N, x, Read (S, p, String M))) =
  if Annotations.IsSQL S                          (* test if S is a relational db *)
  then
   let val Forbidden = SQLDict.ExtractLONG S((!SQL.Mk) M) (* can't migrate tests on BLOB-like fields *)
       val Hash = IntHashTable.mkTable (1, Error.Goofed "Oops!")
       val _ = IntHashTable.insert Hash (x, Forbidden)
       val (Pve, Nve, N') = FlattenTests (CheckPushTest Hash) N
       val Pve = CvtTest S Pve                    (* convert list to tower of if-then-else *)
       val Nve = CvtTest S Nve                    (* convert list to tower of if-then-else *)
   in if (null Pve) andalso (null Nve)
      then NONE
      else SOME (ExtSet (N', x, Read (S, p, String (SQL.PushTest Pve Nve x M)))) end
  else NONE
| PushSelect _ = NONE
```

□

Third is the rule for migrating joins. A special case of this rule is to rewrite `{ y.#mgr | \x <- process "select * from EMP e where 1 = 1" using A, \y <- process "select * from DEPT d where 1 = 1" using A, x.#name = "peter", y.#name = x.#dept }` to `{ y.#mgr | \x <- process "select dept: e.dept from EMP e, DEPT d where e.name = 'peter' and d.name = e.dept" using A }`. In the original query, the entire table `EMP` has to be retrieved once and the entire table `DEPT` has to be retrieved n times, where n is the cardinality of `EMP`. In other words, $n+1$ requests have to be made on the relational database A. In the rewritten query, only one request is made on the relational database A to retrieve the record in the join of `EMP` and `DEPT` matching Peter. More generally, if `x` and `y` are from the same relational database and are always used in the context of a field projection and there are some equality tests `x.#`l_1 `=` `y.#`l_2, then this is a join that we should push to that database. The advantages are that only one request is made, instead of $n+1$; only matching records in the joins are retrieved, instead of entire tables; and the underlying relational database system now also has more context information to perform a better optimization.

Example 6.3 *The rule for migrating joins to a relational database is implemented by* `MigrateJoin` *below. The function* `SQL.PushJoin` *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 MigrateJoin (ExtSet (N, x, Read (S, p, String M))) =
   if Annotations.IsSQL S       (* test if S is a SQL server *)
   then case FullyProjected x N (* test if x is always in a projection *)
     of SOME ProjO =>           (* ProjO are projections on the outer relation *)
     let val Forbidden = SQLDict.ExtractLONG S ((!SQL.Mk) M)
                                (* can't migrate tests on BLOB-like fields *)
         val Hash = IntHashTable.mkTable (10, Error.Goofed "Oops!")
         val _ = IntHashTable.insert Hash (x, Forbidden)
         val (PO, NO, N') = FlattenTests (CheckPushTest Hash) N
                                (* PO, NO are tests on the outer relation
                                   that can be migrated. N' is what's left
                                   after migration *)
     in case CheckJoin Hash x S N'
```

```
      of SOME (PI, NI, ProjI, ExtSet (U, v, Read (_, _, String W))) =>
                               (* W is the inner relation of the join.
                                  PI, NI are tests on W that can be migrated.
                                  ProjI are projections on W. U is what's
                                  left after migration *)
          SOME (ExtSet (Rename x v U, x, Read (S, p, String (
                    SQL.PushJoin x v ProjO ProjI (PO @ PI) (NO @ NI) M W))))
      | _ => NONE end
   | NONE => NONE
  else NONE
| MigrateJoin _ = NONE
```

Most of the work is done by the function `CheckJoin Hash S N'`*, which traverses N' to look for a subexpression that uses the same database S to be the inner relation of the join. If such a relation exists, it returns* `SOME (PI, NI, ProjI, ExtSet (U, v, Read (S, q, String W)))` *such that the set* `W` *can be used as the inner relation of the join and the join condition is* $(\bigwedge \texttt{PI}) \wedge \neg(\bigvee \texttt{NI})$ *and* `ProjI` *stores the projections in which the inner join variable* `v` *occurs and* `U` *is an expression corresponding to the remaining operations that cannot be migrated.* □

Fourth is the migration of selections across two relational databases. An example is the following rewrite of `{ y.#mgr | \x <- process "select * from EMP e where 1 = 1" using A, \y <- process "select * from DEPT d where 1 = 1" using B, x.#name = "peter", y.#name = x.#dept }` to `{ y.#mgr | \x <- process "select dept: e.dept from EMP e where e.name = 'peter'" using A, \y <- process "select mgr: d.mgr from DEPT d where d.name ='" ^ x.#dept ^ "'" using B}`. Here A and B are two different relational databases, so we cannot use `MigrateSelect` to push the test `x.#dept = y.#name` to B. The reason is that B being a database different from A, it has no access to the value of each instance of `x.#dept`. To enable such a migration, the value of each instance of `x.#dept` must be passed dynamically to B, as shown in the rewritten query above where `x.#dept` is dynamically concatenated into the SQL query `select mgr: d.mgr from DEPT d where d.name =` to be passed to B. Note that, in general, `x` does not need to come from a relational database and we simply need to look for equality tests involving the variable of the inner relation that we can migrate.

Example 6.4 *The rule for migrating selections dynamically across two relational databases is implemented by* `MigrateSelectDyn` *below. The function* `SQL.PushTestDyn` *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 MigrateSelectDyn (ExtSet (N, x, Read (S, p, String M))) =
   if Annotations.IsSQL S            (* test if S is a relational database *)
   then
     let val Vars = SyntaxTools.FV N  (* Vars are free variables in N *)
         val Forbidden =  SQLDict.ExtractLONG S ((!SQL.Mk) M)
                                     (* can't migrate BLOB-like fields *)
         val Hash = IntHashTable.mkTable (1, Error.Goofed "Oops!")
         val _ = IntHashTable.insert Hash (x, Forbidden)
         fun Ins y = IntHashTable.insert Hash (y, fn _ => false)
         val _ = IntSet.app (fn y => if y = x then () else Ins y) Vars
         val Vars' = IntSet.difference (Vars, IntSet.singleton x)
                                     (* Vars' are free variables of the entire
                                        expression. If there is a x.#l = y.#l'
                                        for any y in Vars' inside N, then we
                                        may have something to migrate! *)
         val (Pve, Nve, N') = FlattenTests (fn N =>
           (CheckPushTest Hash N) andalso (IntSet.member(SyntaxTools.FV N, x))) N
```

```
                                  (* Pve, Nve are tests that can be migrated
                                     dynamically. N' is what's left. *)
     in if null Pve andalso null Nve
        then NONE
        else SOME (ExtSet (N', x, Read (S, p, Binary (
              fn (X,Y)=> COString.Mk (SQL.PushTestDyn Pve Nve Vars' x (COString.Km X) (CvtTestCO S Y)),
              String M,
              Record (Record.MkTuple (map Variable (IntSet.listItems Vars'))))))) end
  else NONE
| PushSelectDyn _ = NONE
```

The use of the `Binary` (f, E, V) *construct above is notable. When the Kleisli engine encounters this construct, it effectively executes* $f(E, V)$ *using the values of* E *and* V *at that point. In our example,* E *happens to be the original SQL query,* V *happens to store the values on which equality tests are to be performed, and* f *dynamically pushes* V *to* E*!* □

Besides the four rules above, there is also a rule for reordering joins on two relational databases. While we do not provide here the implementation of the reordering rule in Kleisli, let us use an example to explain this optimization. Consider the join `{ y.#mgr | \x <- process "select * from EMP e where 1 = 1" using A, \y <- process "select * from DEPT d where 1 = 1" using B, y.#name = x.#dept }`. We could optimize it as `{ y.#mgr | \x <- process "select dept: e.dept from EMP e where 1 = 1" using A, \y <- process "select mgr: d.mgr from DEPT d where d.name = '" ^ x.#dept ^ '" using B}`. But we could also optimize it as `{ y.#mgr | \y <- process "select mgr: d.mgr, name: d.name from DEPT d where 1 = 1" using B, \x <- process "select 1 from EMP e where e.dept ='" ^ y.#name ^ "'" using A}`. Assume that there is an index on the name field of `DEPT`, then the first optimization is good, because for each `x.#dept`, the cost of the looking up the corresponding manager in `DEPT` from B would be $\log n$ where n is the cardinality of `DEPT`. However, if such an index does not exist, that cost would be n instead of $\log n$. In such a case, if an index happens to exist for the dept field of `EMP` in A, the second optimization would have been much better. More generally, in a nested loop, the order should be swapped if the outer relation is larger than the inner relation and there is an index on the selected field of the outer relation which has good selectivity.

7 A DOE "Impossible" Query

Having seen the optimizations for queries that involve relational database sources, we now show a sample bioinformatics query that benefits significantly from these optimizations. In fact, it is the very first bioinformatics query implemented in Kleisli in 1994 [14], and 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`.) The query was to find for each gene located on a particular cytogenetic band of a particular human chromosome as many of its non-human homologs as possible. Basically, the query means that for each gene in a particular position in the human genome, find DNA sequences from non-human organisms that are similar to it.

In 1994, the main database containing cytogenetic band information was the GDB [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 [10]. 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). The query first extracted the names of genes on the desired cytogenetic band from GDB, and then accessed Entrez for homologs of these genes. Finally, these homologs were filtered to retain the non-human ones. This query was considered "impossible" as there was at that time no system that could work across the bioinformatics sources involved due to their heterogeneity, complexity, and geographical locations. Given the complexity of this query, the CPL query given in [14] was remarkably short. Since then Kleisli has been used to power many bioinformatics applications [4, 12, 11, etc.]

Example 7.1 *The query mentioned is shown below.*[1]

```
sybase-add (#name:"gdb", ...);
readfile locus from "locus_cyto_location" using gdb-read;
readfile eref from "object_genbank_eref" using gdb-read;
{(#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 connect to GDB and map two tables in GDB to Kleisli. After that, these two tables could be referenced within Kleisli as if they were two locally defined sets, `locus` *and* `eref`*. The next 9 lines extract from these tables the accession numbers of genes on Chromosome 22, use the Entrez function* `na-get-homolog-summary` *to obtain their homologs, and filter these homologs for non-human ones.*

Besides the obvious smoothness of integration of the two data sources, this query is also remarkably efficient. On the surface, it seems to fetch the `locus` *table in its entirety once and the* `eref` *table in its entirety n times from GDB, as a naive evaluation of the comprehension would be two nested loops iterating over these two tables. Fortunately, in reality, the Kleisli optimizer is able to migrate the join, selection, and projections on these two tables into a single efficient access to GDB using the optimizing rules from Section 6. Furthermore, the accesses to Entrez are also automatically made concurrent.* □

Since the query above, Kleisli and its components have been used in a number of bioinformatics projects such as GAIA at the University of Pennsylvania (www.cis.upenn.edu/gaia2), TAMBIS at the University of Manchester [4], and FIMM at Kent Ridge Digital Labs [26]. It has also been used in constructing databases in pharmaceutical/biotechnology companies such as SmithKline Beecham, Schering-Plough, GlaxoWellcome, Genomics Collaborative, Signature Biosciences, etc. Kleisli is also the backbone of GeneticXchange Inc. (www.geneticxchange.com).

8 Warehousing of GenPept Reports

Besides the ability to query, assemble, and transform data from remote heterogeneous sources, it is also important to be able to conveniently warehouse the data locally. The reasons to create local warehouses are several: (1) it increases efficiency; (2) it increases availabilty; (3) it reduces risk of unintended "denial of service" attacks on the original sources; and (4) it allows more careful data cleansing that cannot be done on the fly.

The warehouse should be efficient to query and easy to update. Equally important in the biology arena is that the warehouse should model the data in a conceptually natural form. Although a relational database system

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

is efficient for querying and easy to update, its native data model of flat tables forces us to unnaturally and unnecessarily fragment our data in order to fit our data into third normal form.

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

We illustrate this power of Kleisli using the example of GenPept reports. Kleisli provides several functions to access GenPept reports remotely from Entrez [27]: `aa-get-uid-general`, which retrieves unique identifiers of GenPept reports matching a search string; `aa-get-seqfeat-general`, which retrieves GenPept reports matching a search string; `aa-get-seqfeat-by-uid`, which retrieves the GenPept report corresponding to a given unique identifier; and so on. The National Center for Biotechnology Information imposes a quota on how many times a foreign user can access Entrez in a day. Thus, it would be prudent and desirable if we incrementally "warehouse" GenPept reports into a local database.

Example 8.1 *Create a warehouse of GenPept reports. Initialize it to reports on protein tyrosine phosphatases.*

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

In this example, a table `genpept` *is created in our Oracle database system. This table has two columns,* `uid` *for recording the unique identifier and* `detail` *for recording the GenPept report. A LONG data type is used for the* `detail` *column of this table. However, recall from Example 2.2 that each GenPept report is a highly nested complex objects. There is therefore a "mismatch" between LONG (which is essentially a big uninterpreted string) and the complex structure of a GenPept report. This mismatch is resolved by Kleisli which automatically performs the appropriate encoding and decoding. Thus, as far as the Kleisli user is concerned,* `x.#detail` *has the type of GenPept report as given in Example 2.1. So he can ask for the title of a report as straightforwardly as* `x.#detail.#title`*.* □

Normally, when the daily quota for accessing Entrez is exhausted, `aa-get-seqfeat-by-uid` returns the empty set. However, it is possible to configure Kleisli so that a testable "null" value is returned instead. Then it would be useful to regularly examine the local warehouse to update "null" values to their proper GenPept records.

Example 8.2 *A query to check for "null" values in the local warehouse and to replace them with proper GenPept reports. This query should be run regularly whenever new Entrez quota becomes available.*[2]

```
oracle-cplobj-add (#name: "db", ...);
readfile GenPept from "genpept" using db-read;
{ db-update (#table: "genpept", #selector: (#uid: y.#uid), #replacement: (#detail: y))
| \x <- GenPept, x.#detail = null, \y <- aa-get-seqfeat-by-uid (x.#uid)};
```

[2]The `oracle-cplobj-add ...` and the `readfile GenPept ...` part are not needed if the local warehouse is currently connected.

□

It would also be convenient to provide a function `my-aa-get-seqfeat-by-uid` that looks up the local warehouse for GenPept reports first, instead of going straight to Entrez. It would be even more useful if it also automatically updates the warehouse if it ever needs to fetch new reports from Entrez.

Example 8.3 *Implement a "memoized" version of* `aa-get-seqfeat-by-uid`*. It uses the* `before` (E, f) *function of Kleisli which makes sure that the expression E is first evaluated to some value v and then evaluates and returns $f(v)$. Also implement a "memoized" version of* `aa-get-seqfeat-general`*.*

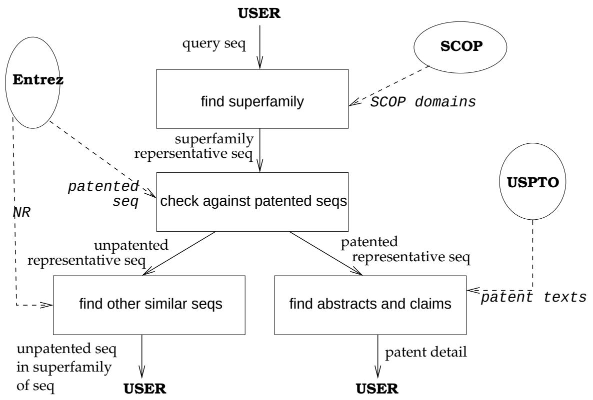
```
oracle-cplobj-add (#name: "db", ...);
readfile GenPept from "genpept" using db-read;
primitive my-aa-get-seqfeat-by-uid == \u => { x.#detail | \x <- GenPept, x.#uid = u} before (\X =>
   if X = { }            ! the desired GenPept report was never fetched
   then { x | \x <- aa-get-seqfeat-by-uid (u),
              _ <- db-insert (#table: "genpept", #replacement: (#uid: x.#uid, #detail: x)) }
   else if X = { null }  ! quota ran out last time we tried to fetch
   then { x | \x <- aa-get-seqfeat-by-uid (u),
              _ <- db-update (#table:"genpept", #selector:(#uid: x.#uid), #replacement:(#detail: x))}
   else X);
primitive my-aa-get-seqfeat-general == \s =>
   { x | \u <- aa-get-uid-general (s), \x <- my-aa-get-seqfeat-by-uid (u) };
```

□

9 A Protein Patent Query

In this section, we demonstrate Kleisli in the context of querying protein patents. We use Kleisli to tie together the following sources to answer queries on protein patents that are considerably more demanding than simple free-text search: (1) the protein section of the Entrez system at the National Center for Biotechnology Information [27]; (2) the BLAST sequence homology service at the the National Center for Biotechnology Information [2]; (3) the WU-BLAST2 sequence homology software from Washington University [1]; (4) the Isite system at the US Patent and Trademark Office (`http://patents.uspto.gov`); and (5) the structural classification of protein database SCOP at Cambridge MRC Laboratory of Molecular Biology [21].

Consider a pharmaceutical company that has a large choice of protein sequences to work on (i.e. 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 figure below.



At the initial stage, we know only the amino acid sequence of these sequences and very little else. A question at this point would be: Which of the sequences have already been patented? Existing patent search systems are IR systems that rely on English words. These systems suffer from the dichotomy of recall vs precision [16]: They either return only the highly relevant information at the expense of missing out a large portion of it, or return most of the highly relevant information at the expense of also returning a lot of irrelevent information. So using the standard interface to patent search systems is laborious and not always fruitful. Furthermore, at this early search, we do not even have an English word to use for searching—we have only the actual amino acid sequences! Thus, we need more reliable technology for comparing our protein sequences to those sequences already patented.

There are, however, two things in our favour. First, protein patents are generally based on protein function *and* primary sequence structure (i.e. the linear string of 20 letters in the amino sequence.) Second, tools that can reliably identify homology at the primary sequence structure level are available [25, 5]. Therefore, if patented protein sequences can be extracted from some database and prepared in a form suitable for such tools to operate on, we will have a means to reliably identify which of our sequences have not yet been patented. We obtain the patented sequences from the protein section of Entrez [27]. These are "warehoused" locally for greater efficiency. We then use WU-BLAST2 [1] for comparing our 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 we do not know the functions of our proteins, because we have not done work on them yet, we know that proteins of the same evolutionary origin tend to have similar functions even if they differ significantly in their primary structure [20]. Using the terminology of SCOP, these proteins are in the same superfamily [21]. So one way to identify protein sequences that have the potential for wider patent claims is to find those having large number of unpatented sequences in the same families. Homology searching algorithms based on primary structure are generally not sufficiently sensitive to detect the majority of sequences in a typical superfamily [25], as the primary structure of distance members of the family are likely to have mutated significantly. So we need tools for homology at the tertiary structure level.

No reliable automatic tools for this purpose exist 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 the SCOP database [21]. Therefore, if we screen our unpatented sequences against SCOP, we can pull out other *representative* sequences in their superfamilies and check which ones have already been patented, thus identifying superfamilies with good potential. We use WU-BLAST2 for 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 BLAST [2] to remotely compare the representatives against the huge nonredundant protein database (NR) curated at the National Center for Biotechnology Information.

Having found these potentially good protein sequences, we are ready to work on them and hopefully patent our results. We thus need to ask the third question: What are the relevant prior arts? Retrieving the texts of patented sequences in the same superfamilies as our proteins would be very helpful here. This step is complementary to the previous step and can be carried out using exactly the same technology.

Example 9.1 *We describe the query to find unpatented sequences in the same superfamily of a user-supplied protein sequence, as it is the most complicated of the three questions we mentioned. The program is shown below.*

```
webblast-blastp(#name: "nr-blast", #db: "nr", #level: 2);             ! line 1
localblast-blastp(#name: "patent-blast", #db: "patent-aa/blast/fasta");
localblast-blastp(#name: "scop-blast", #db: "scop-aa/blast/fasta")
seqindex-scanseq(#name:"scop-index", #index:"scop-aa/seqindex", #level: 1);
scop-add "scop";
readfile scop-summary from "scop-aa/data/summary" using stdin;        ! line 6
primitive scop-accn2uid == (set-index' (#accession, scop-summary)).#eq;
materialize "scop-accn2uid";
{(#title: z.#title, #accession: z.#accession, #uid: z.#uid,
  #class: i.#desc.#cl, #fold: i.#desc.#cf, #superfamily: i.#desc.#sf,
  #family: i.#desc.#fa, #protein: i.#desc.#dm, #species: i.#desc.#sp,  ! line 11
  #scop-pscore: x.#pscore, #nr-pscore: z.#p-n)
| \x <- process SEQ using scop-blast, x.#pscore <= PSCORE-SCOP,
  \i <- process <#sidinfo: x.#accession> using scop,
  \sf <- process <#numsid: i.#type.#sf> using scop,
  \sfuid <- scop-accn2uid (sf),                                        ! line 16
  \y <- process <#get: sfuid.#uid> using scop-index,
  {} = { x | \x <- process y.#seq using patent-blast, x.#pscore <= PSCORE-PATENT },
  \z <- process y.#seq using nr-blast, z.#p-n <= PSCORE-NR };
```

As several data sources are used, we must first connect to them. We establish a connection `nr-blast` *to BLAST at National Center for Biotechnology Information for searching its NR database (line 1). The concurrency level of this connection is set to 2 for more efficient parallel access. We establish a connection* `patent-blast` *to WU-BLAST2 for searching against our local warehouse of patented proteins (line 2). We establish a connection* `scop-blast` *to WU-BLAST2 for searching against our local warehouse of SCOP representative sequences (line 3). We establish a connection* `scop-index` *to the index of SCOP representative sequences (line 4). Both warehouses and the index were constructed previously using Kleisli. We establish a connection* `scop` *to the SCOP classification database (line 5). These different connections to SCOP are needed because the SCOP database (line 5) contains only names and classification but not the actual representative sequences. We keep our sequences using a proprietary sequence indexing technology SeqIndex [23]. This index (line 4) allows us to quickly retrieve a sequence given an identifier or a pattern. Unfortunately, WU-BLAST2 requires the sequences to be stored in a different format. Hence we need the third connection to SCOP (line 3). We must also deal with one more problem: The identifiers used by SCOP (lines 3, 5) are different from the identifiers used by SeqIndex (line 4). We therefore need a map between these identifiers, which is captured in the relation* `scop-summary` *(line 6). For quick access we create a memory-resident index* `scop-accn2uid` *to map SCOP identifiers to SeqIndex identifiers (lines 7-8).*

After setting up connections to various data sources as described above, we are ready to issue our query to retrieve information about unpatented sequences in the same superfamily as our protein sequence `SEQ`*. The information returned include title, accession, unique identifer, and classification of these sequences (lines 9–11). Also returned with each unpatented protein sequence is its pscore with respect to SCOP and NR (line 12). The pscore is a reliable estimate of the corresponding sequence being a false positive, given by BLAST and WU-BLAST2 [2, 1]. Eg., if*

BLAST returns a hit with pscore of 0.001 to your sequence, then there is a one in a thousand chance of the hit being a fluke.

Let us step through the body of the program. Given the user sequence `SEQ`*, we compare it against representative sequences in SCOP; we keep only those hits* `x` *whose pscore is within the error threshold* `PSCORE-SCOP` *(line 13). Since each hit* `x` *is good, the superfamily of* `x` *can be taken as the superfamily of the input sequence* `SEQ`*. We find the superfamily of* `x` *by simply asking SCOP to return us the SCOP classification* `i` *of* `x` *(line 14). The name and identifier of* `x`*'s superfamily is stored in the* `#desc.#sf` *and* `#type.#sf` *fields of* `i` *respectively. Next, we need to fish out all representatives of that family from SCOP. SCOP gives us their SCOP identifiers* `sf` *(line 15). We convert these identifiers into unique identifiers* `sfuid` *in the SeqIndex where the sequences are kept (line 16). The SeqIndex is then accessed to give us the actual representative sequences* `y` *(line 17). Each representative sequence* `y` *is compared against patented sequences. We retain those that have no hits within the error threshold* `PSCORE-PATENT` *(line 18). These are representatives of our superfamily that are dissimilar to every patented sequence. We compare them against the NR database to fish out all sequences* `z` *that are similar to them within the error threshold* `PSCORE-NR` *(line 19). These are all the desired unpatented sequences in our superfamily.* □

Although the details of this process may seem confusing and requires knowledge of the biomedical data resources available, it should be clear that a high-level technology such as Kleisli [32, 33, 14] greatly simplifies the process of developing interesting integrated systems. Furthermore, the Kleisli/CPL programs that access multiple distributed databases and softwares are clear and concise, easily written and easilty modifiable. The ability to return information on protein patents as shown above goes well beyond the reach of existing patent information servers based on standard IR systems.

10 A Clinical Data Query

We discuss one last example application of Kleisli; this time, in the context of querying clinical data. Clinical data usually involve historical records of patient visits. Queries over such data often involves the concept of "window", which is difficult to express using SQL.

We illustrate this point using a clinical database of Hepatitis-B patients. The database has a table `BLD_IMG` of type `{(#patient: string, #date: num, #ALT: num, ...}`.[3] Each time a patient's ALT level is measured, a new record is added to this table. One typical query of this data might be: "Find patients who have been tracked for at least 300 days and whose ALT level is always between 70 and 200 units." Such a query is easily expressed in SQL using the `GROUP-BY` and `HAVING` constructs and the aggregate functions `MIN` and `MAX`. However, another typical query might be: "Find patients whose ALT levels have stayed between 70 and 200 units for a period of at least 300 days." This query is problemmatic for SQL.

The challenges posed by the second query are the followings. First, the records must be grouped by patients. Second, the records in each group must be sorted in chronological order. Third, the sorted records in each group must be segmented into subgroups such that within each subgroup either all ALT levels are between 70 and 200 units or all the ALT levels are outside this range; futhermore, the date span of each subgroup must not overlap another subgroup. Fourth, return the records of a patient if he has a subgroup that spans at least 300 days and all ALT levels in that subgroup are between 70 and 200 units. The first, second, and fourth steps are straightforward in SQL. Unfortunately, the third step is not do-able in SQL.

This query is also not expressible in the Kleisli system using just the core calculus $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, \leq^{\mathbb{Q}}, =)$. However, after a combinator for structural recursion [6] is imported into Kleisli, the query becomes expressible. The combinator is `list-sri`, which corresponds to the `fold` operation in functional programming languages.

[3]As we do not want to complicate our discussion with arithmetics on dates, we treat the value of the `date` attribute in our patient database as an integer corresponding to the number of days passed since a particular fixed time point.

Example 10.1 *We can define a function* `split` *in terms of* `list-sri` *so that* `split` (d, c) S *returns a list* $[(x_1, Y_1), \ldots, (x_n, Y_n)]$ *such that* $Y_1, \ldots, Y_n$ *is* S *sorted chronologically;* $x_1, \ldots, x_n$ *alternate between true and false; and* $c(y_j) = x_i$ *for each* $y_j \in Y_i$. *In the implementation below,* `s2l` *converts a set into a list and* `list-gensort` *sorts a list using a given ordering.*

```
primitive split == (\date, \check) =>
   (list-sri ((\x, \y) =>
      if y = []
      then [ (x.check, [x]) ]
      else if y.list-head.#1 = x.check
           then (y.list-head.#1, x +] y.list-head.#2) +] y.list-tail
           else (x.check, [x]) +] y, [])) o
   (list-gensort ((\x, \y) => x.date > y.date)) o s2l;
```

Then the query "Find patients whose ALT levels have stayed between 70 and 200 units for a period of at least 300 days" can be defined as follows.

```
! access the hepatitis-B database on Oracle
oracle-cplobj-add (#name: "hepB", ...);
readfile BLD_IMG from "BLD_IMG" using hepB-read;
! define the ``window'' based on ALT level
primitive window == split (#date, \x => (x.#ALT > 70) andalso (x.#ALT <= 200));
! compute the answer
{ p | \k <- set-unique { x.#patient | \x <- BLD_IMG },
      \P == { y | \y <- BLD_IMG, y.#patient = k },
      {()} = { () | \g <--- window (P), g.#1,
                    (g.#2.list-rev.list-head.#date - g.#2.list-head.#date) >= 300},
      \p <- P };
```

Incidentally, this implementation also demonstrates the seamless integration of the operation of Kleisli, such as the use of "window", with that of a relational database. □

11 Conclusion

The Kleisli system and its high-level query language CPL embody many advances that have been made in database query languages and in functional programming. It represents a substantial 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 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 succinctly as Kleisli.

There are several key ideas behind the success of the system. The first is its use of a complex object data model where sets, bags, lists, records and variants can be flexibly combined. The second is its use of a high-level query language (CPL) which allows these objects to be easily manipulated. The third is its use of a self-describing data exchange format, which serves as a simple conduit to external data sources. The fourth is its query optimizer, which is capable of many powerful optimizations. The last-but-not-least reason behind the success of the system is the choice of SML as its implementation platform, which enables a remarkably compact implementation consisting of about 45000 of codes in SML. We have no doubt that without this robust platform of functional programming, it would have demanded much more effort to implement Kleisli.

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

References

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

[2] S. F. Altschul et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

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

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

[5] G. J. Barton and M.J.E. Sternberg. Evaluation and improvements in the automatic alignment of protein sequences. *Protein Engineering*, 1:89–94, 1987.

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

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

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

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

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

[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. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 593–595, June 1998.

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

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

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

[16] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

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

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

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

[20] H. Lodish et al. *Molecular Cell Biology*. W. H. Freeman, New York, 1995.

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

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

[23] H.-H. Pang et al. S-Hash: An indexing scheme for approximate subsequence matching in large sequence databases. Technical report, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, 1997.

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

[25] W. R. Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4:1145–1160, 1995.

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

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

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

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

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

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

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

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