A Data Exchange Format and Some Supporting Tools

Jing Chen Limsoon Wong*
BioInformatics Centre & Kent Ridge Digital Labs
21 Heng Mui Keng Terrace, Singapore 119613
{cjing,limsoon}@krdl.org.sg

14 September 1998

Abstract

We describe the concept of an open query system based on a nested relational data model and a self-describing data exchange format. The Kleisli Query System is such an open query system. We present the data model and exchange format of the Kleisli Query System. Many data sources in bioinformatics can be easily mapped onto them. Then we describe Kleisli/CPL and Kleisli/Perl, which are two tools for manipulating objects conforming to the data model and exchange format, through an extended example on Kozak sequences. The example also shows that it is simple to use the Kleisli Query System to perform fairly complex integration, transformation, and analysis.

1 A Family of Query Systems

Many problems in bioinformatics involve: accessing multiple data sources that are complex, heterogeneous, and geographically dispersed; manipulating and re-structuring of data derived from various databases and analysis programs; and passing data smoothly between multiple analytical steps. Simple retrieval of data is not likely to meet the demand of modern bioinformatics. Kleisli/CPL [9] was one of the first general bioinformatics integration systems that went significantly beyond simple linking and retrieval. It made some impression on the field[4, 14, 2, 3] as a powerful system for implementing general database-style integration and transformation of bioinformatics sources.

However, the use of Kleisli/CPL has now gone beyond database-style integration. Increasingly, it has also been used to program certain kind of tasks that are best left for string-oriented languages such as Perl, or graphical-user-interface-oriented languages such as Java, etc. We feel that it is now time to expand Kleisli/CPL into a family of query systems, each specialized for the support of tasks of different nature, and each allowing the programming to be performed using the most suitable language for that task. However, we also want to make sure that any combination of members of such a family of query systems can be used together easily, as if they are practically a single unified system.

The objective of this paper is to describe a way to build such a family of query systems that will collectively be viewed as the Kleisli Query System.[1] There are three central ideas. The first is that of simple structural data model rich enough to model all structural aspects of data used by every query system in the family. The second is that

*Contact author.

[1]We plan to rename it to KRIS, which stands for both "Kleisli-Related Integration System" and "Kent Ridge Integration System" to distinguish it from the early prototype developed several years ago at the University of Pennsylvania. This new name KRIS was chosen also because it means the "wavy-bladed" type of daggers original to the Malay archipelago where the industrial-strength version of Kleisli was developed—Singapore.

of a simple self-describing data exchange format for representing data conforming to the data model in a way that is independent of any query system in the family and independent of any programming languages. The third is the carefully thought out incorporation of the data model and exchange format into the query systems of the family and their programming languages.

We explain our rationale. We insist on a *structural* data model. That means all we care about, when given a piece of data, is its structural aspect—it is a list? a record? or a number?—what the list, record, or number actually means at the level of *application semantics* is left to the application programmer. This makes it very easily to map the data model into any modern programming languages. We insist on a *self-describing* exchange format. That means a data stream laid out according to this format must be easy to parse and must not depend on any schema or dictionary definitions that must be exchanged beforehand. This makes it very easy to develop interface to this exchange format for any modern programming languages. We insist on incorporating the data model and exchange format into every query system of the family. This makes it easy to share and pass data between different members of the family, allowing them to be used together in solving problems.

These simple ideas go quite a long way. In the main body of the paper, we describe the realization of them in the Kleisli Query System. Section 2 presents the data model and exchange format of the Kleisli Query System. Section 3 introduces Kleisli/CPL via an example of the extraction of context sequences of translation initiation sites of *Arabidopsis* signal transduction-related genes. Section 4 introduces Kleisli/Perl, a query system within the Kleisli family that is more suited to string manipulations. We will use it to calculate the weight matrix and consensus of our context sequences as a demonstration. Section 5 provides some closing remarks, as well as a brief comparison of the Kleisli Query System to three representatives related work.

2 A Data Exchange Format

A key idea in the Kleisli Query System is its data exchange format that supports a rich complex object data model. This section gives a brief introduction to this data model and exchange format.

The data model of Kleisli is a complex object type system that goes beyond the "sets of records" type system of relational databases. It allows arbitrarily nested records, sets, lists, bags, and variants. A bag is also called a multiset, which is conceptually a set in which duplicates may occur. A list is conceptually a bag with order. A variant is also called a tagged union and represents a type that is "either this or that" [12]. The sets, bags, and lists in the Kleisli Data Model are assumed to be homogeneous. So 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. However that the Kleisli Data Exchange Format (described below) does not insist on sets, bags, and lists being homogeneous.

As an example of a complex type expressible in this data model, consider the following type that represents the feature table of a GenPept report from Entrez [17, 20].

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

It has 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 paritcular 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 in the data model are freely formed by the syntax:

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

Here `num`, `string`, and `bool` are the base types. The other types are constructors and build new types from existing types. $\texttt{\{}t\texttt{\}}$, $\texttt{\{|}t\texttt{|\}}$, and $\texttt{[}t\texttt{]}$ respectively construct set, bag, and list types from type t. $\texttt{(}l_1 : t_1, ..., l_n : t_n\texttt{)}$ construct record types from types $t_1, ..., t_n$. $\texttt{<}l_1 : t_1, ..., l_n : t_n\texttt{>}$ construct variant types from types $t_1, ..., t_n$.

To allow any external system to exchange data with the Kleisli Query System, a data exchange format that corresponds to Kleisli's data model is specified. The Kleisli Data Exchange Format is a self-describing format[2] for laying out complex objects in a data stream. It has a basic form and a compressed form. The important property of the format is that each data type is given an unambiguous encoding. As a result, the data items can be parsed without reference to any schema. In fact, it is designed so that no more than two characters need to be looked at to know what type of data is being transmitted.

The basic form of the exchange format is given in the table below. Punctuation/indentation marks such as colon, commas, tabs, newlines, and spaces are not significant. The semicolon is used to indicate the end of a complex object.

Data Type	Data Layout	Remarks
Unit	`()`	
Booleans	`true` `false`	

Numbers	`123`
	`123.123`
	`~123`
	`~123.123`
		Positive numbers
		
		Negative numbers

Strings	`"a string"`	String is put inside double quotes
Records	$\texttt{(\#}l_1\texttt{:}\ O_1,$ $\vdots,$ $\texttt{\#}l_n\texttt{:}\ O_n\texttt{)}$	Record is put inside round brackets. The label-value triplets enumerate the fields of the record
Variants	$\texttt{<\#}l\texttt{:}\ O\texttt{>}$	Variant is put inside angle brackets
Sets	$\texttt{\{}\ O_1,$ $\vdots,$ $O_n\texttt{\}}$	Set is put inside curly brackets
Bags	$\texttt{\{|}\ O_1,$ $\vdots,$ $O_n\texttt{|\}}$	Bag is put inside curly-bar brackets
Lists	$\texttt{[}\ O_1,$ $\vdots,$ $O_n\texttt{]}$	List is put inside square brackets
User-defined types	`longitude "50E"`	User-defined type is preceded by its name
Errors	`error "it goofed"`	Error message is preceded by `error`

Here is an example layout of a complex object:

```
{(#history: {},
```

[2]Self-describing data exchange formats exist in several forms in earlier work [19, 22, 16, etc.] However, Kleisli's is probably the first self-describing exchange format that is consciously derived from type inference! [23]

```
  #name: "jim"),
 (#history: {(#salary: 2100, #date: "july 1993"),
             (#salary: 2000, #date: "june 1993")},
  #name: "tom")}
```

Here is the same object laid out in an equally acceptable way:

```
{(#history: {}, #name: "jim"),
 (#history: {(#salary: 2100, #date: "july 1993"),
 (#salary: 2000, #date: "june 1993")}, #name: "tom")}
```

The compressed form of the exchange format adds a few shorthands to the basic form. Basically it allows the definition and use of shorthands. There are three categories of shorthands: labels, records, and user-defined types. In each category, upto a maximum of 255 shorthands can be defined simultaneously within a data stream. However, any shorthand can be redefined at any time within a data stream.

The shorthand for a long label is defined by replacing an occurrence of the label `#`l in the data stream by `|DEFL`c `#`l. Note the space between c and the label. Here c is any single character except the newline character. From that point onward, all occurrences of `|L`c are interpreted as `#`l. For example, the earlier complex object is equivalent to the one below.

```
{(|DEFL1 #history: {}, |DEFL4 #name: "jim"),
 (|L1: {(|DEFL2 #salary: 2100, |DEFL3 #date: "july 1993"),
 (|L2: 2000, |L3: "june 1993")}, |L4: "tom")}
```

The shorthand for a record is defined by replacing an occurrence of a record `(#`l_1`:` O_1, ..., `#`l_n`:` O_n`)` in the data stream by `|DEFR`, followed by c, followed by a space, followed by the field labels of the record separated by single spaces, followed by `(`, followed by the field values listed in the same order as the field labels, followed by `)`. That is, `|DEFR`c `#`l_1 ... `#`l_n `(` O_1, ..., O_n `)`. Here c is any single character except the newline character. From this point onward, all occurrences of `|R`c `(` O'_1 ... O'_n `)` are interpreted as the record `(#`l_1`:` O'_1, ..., `#`l_n`:` O'_n`)`. For example, the complex object above is equivalent to the following:

```
{ |DEFR1 #history #name ({}, "jim"),
  |R1 { |DEFR2 #salary #date (2100, "july 1993"),
        |R2 (2000, "june 1993")}, "tom")}
```

This compressed form is especially useful for transmitting relational tables. For instance, the table

```
{ (#name: "abel", #age: 20, #city: "Alaska"),
  (#name: "carol", #age: 10, #city: "California"),
  (#name: "david", #age: 40, #city: "Dakota"),
  (#name: "francis", #age: 22, #city: "Florida")}
```

can be laid out more compactly as

```
{ |DEFR1 #name #age #city ("abel", 20, "Alaska"),
  |R1 ("carol", 10, "California"),
  |R1 ("david", 40, "Dakota"),
  |R1 ("francis", 22, "Florida")}
```

The shorthand for a user-defined type is defined by replacing an occurrence of the type name N by `|DEFT`c N, where c is any single character except the newline character. From this point onward, all occurrences of `|T`c in the data stream are interpreted as N. Finally, note that in order to avoid confusing lexical analysers that use only single-character lookaheads between `{|` and `{|R` (`{|DEFR`, etc.), it is also required that a space be inserted between the `{` and `|R` (`|DEFR`, etc.)

Now we return to our example on GenPept feature tables from the beginning of this section. A value conforming to our feature table type is laid out in the Kleisli Exchange Format like this:

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

This is a rich data model. The schemas and structure of all popular bioinformatics databases, flat files, and softwares are easily mapped into it. `//adenine.krdl.org.sg:8080/publications/drivers.html` is a partial list of external sources that the Kleisli Query System talks to via this exchange format.

3 Kleisli/CPL in Action

For a data model or an exchange format to be useful, it must be supported by an adequate manipulation mechanism. The Collection Programming Language (CPL) is designed for high-level database-style manipulation and transformation of data expressed in the Kleisli Data Model. It incorporated many advances in query language theory such as parametric polymorphism, comprehension syntax, higher-order function, query optimization, etc. A complete description of our implementation of CPL in the context of the Kleisli Query System is available [24]. An overview of an older version is also available [9]. So in this section, we just provide an extended example for illustration.

Our example is this: we would like to extract Kozak sequences of *Arabidopsis* genes that encode proteins involved in signal transduction and regulation. Kozak sequences are the flanking regions of translation initiation sites[15]. We outline the steps to accomplish this task so that it is easier appreciate how Kleisli/CPL can simplify the life of a bioinformatics programmer. In the interest of space, our program is slightly simplified.

1. We have to obtain *Arabidopsis* DNA sequences that have explicitly annotated translation start sites. This involves downloading sequences and their feature tables from NCBI Entrez and is accomplished by the CPL program below. (Line numbers are added for easy reference.)

   ```
   1.   writefile
   2.     { (#translation: a.#descr, #kozak: z)
   3.     | \u <- na-get-uid-general "Arabidopsis[Organism] AND cds[Feature key]",
   4.       \s <- na-get-seq-by-uid (u), \t <- na-get-feature-by-uid (u),
   5.       \f <- t.#feature, f.#continuous, f.#name = "CDS",
   6.       \p == f.#position.list-head,  not (p.#negative),
   7.       \a <--- f.#anno, a.#anno_name = "translation", a.#descr string-islike "M%",
   8.       \z == string-span (s.#sequence, p.#start - 15, p.#start + 12),
   9.       string-length (z) = 28, string-span (z, 16, 18) = "ATG" }
   10.  to "kozak" using stdout;
   ```

   This CPL program creates a set of Kozak sequences and the translation of their corresponding proteins and writes it to a file "`kozak`". Here is how it works. Line 3 obtains from Entrez numbers `u` that uniquely identify *Arabidopsis* DNA sequences in GenBank that have explicitly annotated coding regions (CDS). Line 4 downloads from Entrez GenBank feature tables `x` and corresponding sequences `s` using identifiers `u`. Line 5 extracts complete CDS entries `f` from feature tables of reports `x`. Line 6 extracts positions of first exons from these CDS entries `f`, provided they are on the positive strand. Line 7 extracts annotations `a` associated with these CDS; it makes sure that these annotations are the translation of these CDS and that these translations begin with the expected `"M"`. Line 8 extracts the -15 to +12 flanking nucleotides surrounding the start of these CDS, these are our Kozak sequences `z`. Line 9 performs the obvious error control. The extracted translation `a.#descr` and Kozak sequences `z` are put into a record (Line 2) and written out.

2. In order to select from Step 1 those Kozak sequences that belong to genes encoding signal transduction and regulation proteins, it is actually possible to modify Line 3 of the CPL program above to something like

```
3a.  | \u <- na-get-uid-general "Arabidopsis[Organism] AND cds[Feature key] AND SPEC",
```

where `SPEC` is a suitable combination of keywords that describe what it means to be a gene involved in signal transduction and regulation.[3]

However, we need to deal with a problem: the functional annotations in GenBank are not very complete. So doing the above we are going to miss quite a few *Arabidopsis* signal transduction and regulation genes that lack these annotations. To increase our chance of catching them and also to make our example more interesting, we can do this. We first use the appropriate keywords to download signal transduction and regulation *protein* sequences of seed plants (*Spermaphyta*). Then we run BLASTP [1] to compare the translations from Step 1 against them. As sequences sharing high sequence homology are likely to have the same function, doing it this way we should be able to use the seed plant signal transduction and regulation proteins to fish out a more complete set of relevant Kozak sequences of *Arabidopsis* genes.

So our Step 2 is the following CPL program that downloads the relevant seed plant protein sequences from Entrez and builds a BLASTP database `"signal-blast"`.

```
11. writefile
12.   {(#uid: x.#uid, #accession: x.#accession, #title: x.#title, #seq: x.#sequence)
13.   | \u <- aa-get-uid-general "Spermaphyta[Organism] AND SPEC",
14.     \x <- aa-get-seq-by-uid (u)}
15. to "signal-blast" using localblast-setdb;
```

3. Now we simply perform BLASTP for each of the protein translation from Step 1 against the BLAST database constructed in Step 2. Those that have a strong hit (pscore $< 10^{-6}$) are retained and the corresponding Kozak sequences are written to a file `"signal-kozak"` after duplicates are removed. Here is the CPL program.

```
16. readfile kozak from "kozak" using stdin;
17. localblast-blastp (#name: "signal-blast", #db: "signal-blast");
18. writefile set-unique
19.   { x.#kozak
20.   | \x <- kozak,
21.     \H == { h | \h <- process x.#translation using signal-blast, h.#pscore < 1E~6},
22.     not (set-isempty (H))}
22. to "signal-kozak" using stdout;
```

4 Kleisli/Perl in Action

The high-level query language, CPL, of the Kleisli Query System was designed to express traditional (nested relational) database-style queries. Not every query in bioinformatics falls into this class. For these non-database-style

[3]See `http://alanine.krdl.org.sg:8080/ limsoon/ kozak/ arabidopsis/ wls-signal-transduction` for the definition of `SPEC` we used.

queries, some other programming languages can some time be a more convenient or more efficient means of implementation. Following the vision described in Section 1, we are interfacing a family of programming languages to the data exchange format of Kleisli. In this section, we describe Kleisli/Perl, the interface between Kleisli and Perl.

In contrast to Kleisli/CPL, which is a high-level interface that comes with a sophisticated optimizer [25] and other database-style features, Kleisli/Perl is has a different purpose and is at a lower level. Whereas Kleisli/CPL is aimed at extraction, integration, and preparation of data for analysis, Kleisli/Perl is intended to be used for implementing analysis and textual formatting of the prepared data. Thus, we designed Kleisli/Perl as a Perl module for parsing data conforming to the data exchange format of Kleisli into native Perl objects.

We describe the packages in Kleisli/Perl before demonstrating it on our running example. There are three packages in Kleisli/Perl:

1. The RECORD package simulates the record data type of our data exchange format by using a reference of Perl's hash. Three functions are defined in this package:

   - `new` is the constructor of a record. It works the same way as you would initialize a hash. For example, to create a record such as `(#name: "tom", #age: 12)` in your Perl program, you can write:

     ```
     $rec = RECORD->new ( "name", "tom", "age", 12);
     ```

     where `$rec` will be the reference of this record in your Perl program.

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

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

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

   - `HasField` checks whether the specified field is exist in a record. For example

     ```
     $rec->HasField("age");
     ```

     will return a non-zero value if and only if the record referenced by `$rec` in your Perl program has the `#age` field.

   We should remark also that the variant data type in the Kleisli Data Exchange Format is also simulated in Kleisli/Perl using the RECORD package. Basically a variant object from the data exchange format is simulated in Kleisli/Perl as if it is a record object having exactly one field having exactly the same name as the tag of the variant. This simulation of the variant data type is reasonable compromise as Perl is not strongly typed and lacks the equivalent of variant.

2. The LIST package simulates the list, set, and bag data type in the Kleisli Data Exchange Format. All these three "bulk" data types will be converted and stored as a reference of Perl's list. It has only one function:

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

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

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

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

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

     ```
     $hd = CPLIO->Open( IN, "person.val");
     ```

     or even this (where the same file is piped through the Unix "cat" command):

```
$hd = CPLIO->Open( IN, "/usr/bin/cat person.val |");
```

In both cases, `$hd` will have the handle of the opened file in Perl. Similar to the usage of the "open" operation in Perl, Our `Open` also needs the user to provide a constant to indicate the input stream, which is `IN` in this example.

- `Open1` is a convenient variation of `Open`. `Open1` only takes one parameter and it can used in this way:

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

- `Open1a` is another version of `Open` function which can take a string as input stream. The first parameter specified the child process you want to execute and the second parameter is input string. Here is an example:

```
  $cmd = qq{
  readfile X from "/usr242/BIC/perl-cjing/machperson" using stdin;
  X;
};
$a= CPLIO->Open1a ( "/usr/kleisli/bin/cpl4perl" , $cmd);
```

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

```
($a, $b) = CLPIO->Open2("/usr/kleisli/bin/cpl4perl");
print $b $cmd1;
flush $b;
$a->Parse;
...
print $b $cmd2;
flush $b;
$a->Parse;
...
```

- `Close` is the function to close a file opened by `Open`. For example:

```
$hd->Close;
```

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

```
$set = $hd->Parse;                #get the whole set
foreach $rec (@{$set} ) {         #iterate over each record
  $n = $rec->Project( "name");    #get name of current record
  print "$n\n"; }                 #print it
```

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

```
while (1) {
  $rec = $hd->LazyRead;           #get next record
  if ($rec eq "") last;           #if no more then quit
  $n = $rec->Project( "name");    #else get name current record
  print "$n\n"; }                 #and print
```

- `EOF` is a function that returns true when "end of file" is reached.

Having seen the main functions of the packages of Kleisli/Perl, let us use the remainder of this section to briefly demonstrate one of them—CPLIO. In the previous section we generated a file `"signal-kozak"` of Kozak sequences of *Arabidopsis* signal transduction and regulation genes. We use Kleisli/Perl to compute a weight matrix for them.

```
23. $fd= CPLIO->Open (IN, "signal-kozak.val");
24. $total = 0;
25. while !($fd->EOF) {
26.  $seq = $fd->LazyRead;
27.  ++$total;
28.  $i = 0; while ($i < 28) {
29.           $c = substr $seq, $i, 1;
30.           ++$HASH{$c}[$i];
31.           ++$i; } }
32. $i = 0; while ($i < 28) {
33.          $ap = ($HASH{A}[$i] / $total) * 100;
34.          $cp = ($HASH{C}[$i] / $total) * 100;
35.          $gp = ($HASH{G}[$i] / $total) * 100;
36.          $tp = ($HASH{T}[$i] / $total) * 100;
37.          print "$i:  $ap  $cp  $gp  $tp\n";
38.          ++$i; }
```

Line 23 opens the `"signal-kozak"` file for reading (the `writefile` command of Kleisli/CPL added the `".val"` extension to the file name.) Then we begin looping through Lines 25 to 30. At each iteration, Line 26 lazily reads the next Kozak sequence, while Lines 28 to 31 update the weight matrix. Once we finish with this loop, we begin a second loop through Lines 32 to 38. This second loop converts the weight matrix from frequency count to percentages and prints it out.

For completeness, the profile of the 686 nonredundant Kozak sequences of *Arabidopsis* genes involved in signal transduction and regulation obtained in our example is displayed below. We also display their consensus calculated according to the usual 50/75 rule [6], which is very consistent with that of *dicot* genes in general[13].

position	-12	-13	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	ATG	4	5	6	7	8	9
A%	38	38	38	33	38	38	40	35	50	53	43	46	-	18	27	21	33	30	26
C%	17	16	17	16	22	17	16	25	14	11	28	21	-	7	40	13	16	31	16
G%	17	18	16	20	15	19	21	17	19	23	8	19	-	58	18	31	28	17	24
T%	27	28	28	31	24	26	23	23	17	12	22	14	-	16	15	35	24	22	33
consensus	a	a	a	a	a	a	a	a	a	A	a	a	ATG	G	c	t	a	c	t

5 Closing Remarks

We discussed a plan of a family of query systems that shared a common general data model that was purely structural and exchanging their data via a common easy-to-process self-describing exchanging format. Then through a series of examples, we defined the data model, exchange format, and query languages for their manipulation that collectively formed the Kleisli Query System. We recap the main points here:

- A data model where structures such as records, variants, sets, bags, and lists can be mixed was then proposed. A self-describing exchange format for representing and exchanging data conforming to this data model in a linear character stream was also specified. We also showed through the example of GenPept feature tables how to cast complex bioinformatics data into this data model.

- A high-level query language, CPL, was briefly introduced as the primary database-style "interface" to the Kleisli Data Model. We saw that a very short CPL program (22 lines long) was sufficient to obtain Kozak sequences of *Arabidopsis* genes that were involved in signal transduction and regulation. The salient point here was that using nothing more than database-style CPL queries in Kleisli we easily integrated many things: access to DNA and protein sequences and GenBank reports from Entrez, extract feature annotations from GenBank reports, construction of BLAST databases, performance of BLASTP searches on them, and extraction and manipulation of BLAST results.

  There was also an aspect that was not obvious from the above: optimization and parallelization. Take Line 4, where we downloaded the corresponding GenBank entry `x` for each *Arabidopsis* identifier `u` for instance. Kleisli/CPL did not download them one after another. Kleisli/CPL recognized that Entrez was located far away and that there was a lot of network latency and that the download of different `x` for different `u` were independent. Thus Kleisli/CPL actually downloaded these `x` several at a time, up to the level of concurrency sustainable by the network and by Entrez.

- A lower level interface, Kleisli/Perl, was described next. It provided enough facility to manipulate data conforming to the data model and exchange format of the Kleisli Query System within Perl. It allowed us to take advantage of Perl's special capability in string processing to program interesting analysis and display routines. In particular, we used Kleisli/Perl to construct a profile of our Kozak sequences obtained earlier. This extended example also showed that Kleisli/Perl could postprocess results created by Kleisli/CPL.

There are more interesting ways of using Kleisli/CPL and Kleisli/Perl together to solve bioinformatics problems, such as our Protein Patent Query System [8], but we do not have room in this paper to illustrate them. We also have plans to develop interfaces between other programming languages and Kleisli, the prime targets being C, Java, Pizza [18], GJ[5], and Haskell[11]. The first two are chosen because of their popularity; the last three because of their status as interesting emerging research languages.

We end this paper by making a brief comparison to three previous systems.

- It is fair to say that SRS [10] is one of the more successful tools for integrating biology databases. It is essentially a link-based interface. It is very convenient to use for simple operations. However, it offers no facility for flexible transformation of data from these sources, an important aspect in more advanced bioinformatics applications. As a consequence, one needs a significant amount of manual work if one relies mostly on SRS for data integration.

- TAMBIS [3] is slightly more comparable to Kleisli. It is an interface to many sources that come with a rather fixed set of queries that offer more slightly interesting integration of the sources than SRS. The kind of queries that can be expressed basically have to fall into templates anticipated by TAMBIS' designers. Thus, the expressible queries are much more limited than in Kleisli/CPL. In compensation to the limitation on its query expressive power, its queries can be formulated using a looser syntax and is supported by a knowledge-driven user interface. This could be an advantage for less skillful programmers. However, it is worth pointing out that TAMBIS actually runs on top of an old version of Kleisli/CPL, demonstrating the value of the Kleisli System as a substrate for constructing other integration systems.

- Closest to Kleisli is the OPM approach [7], which also offers a data model and a single high-level database-style query language but without an accompanying simple exchange format. The data model has more semantics than Kleisli's purely structural ones. The extra semantics could be exploited for more optimization and better graphical interfaces, but it also implies less flexibility and demands more planning and work in order to cast an external data source into it. The lack of a simple exchange model further compounds this disadvantage. To overcome the original OPM's inability to easily exchange data with other sources, they recently adopted the heavy weight solution of exchanging data through CORBA [21]. CORBA is a considerably more complex solution than Kleisli's simple self-describing exchange format. In addition, while relational databases fit well into OPM, the situation is not as clear for more sophisticated and specialized bioinformatics softwares.

References

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

[2] P. G. Baker and A. Brass. Recent development in biological sequence databases. *Current Opinion in Biotechnology*, 9:54–58, 1998.

[3] P. G. Baker, A. Brass, S. Bechhofer, C. Goble, N. Paton, and R. Stevens. TAMBIS—transparent access to multiple bioinformatics information sources. In *Proceedings of 6th International Conference on Intelligent Systems for Molecular Biology*, pages 25–34, June 1998.

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

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA 98*, October 1998. To appear.

[6] D. R. Cavener. Comparison of the consensus sequence flanking translation start site in *Drosophila* and vertebrates. *Nucleic Acids Research*, 15:1353–1361, 1987.

[7] I.-M. A. Chen and V. M. Markowitz. An overview of the object-protocol model (OPM) and OPM data management tools. *Information Systems*, 20(5):393–418, 1995.

[8] J. Chen, L. Zhang, and L. Wong. A protein patent query system powered by Kleisli. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 593–595, June 1998.

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

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

[11] J. H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. The functional programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[12] R. Hull and C. K. Yap. The Format model: A theory of database organisation. *Journal of the ACM*, 31(3):518–537, July 1984.

[13] C. P. Joshi, H. Zhou, X. Huang, and V. L. Chiang. Context sequences of translation initiation codon in plants. *Plant Molecular Biology*, 35:993–1001, 1997.

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

[15] M. Kozak. An analysis of 5'-noncoding sequences from 699 vertebrate messenger RNAs. *Nucleic Acids Research*, 15:8125–8148, 1987.

[16] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[17] J. McEntyre. Linking up with Entrez. *TIG*, 14(1):39–40, January 1998.

[18] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[19] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogenous information sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

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

[21] J. Siegel. *CORBA: Fundamentals and Programming.* Wiley, New York, 1997.

[22] G. Wiederhold. *File Organization for Database Design.* McGraw-Hill Book Company, New York, 1987.

[23] L. Wong. An introduction to the Kleisli query system and a commentary on the influence of functional programming on its implementation. *Journal of Functional Programming.* To appear.

[24] L. Wong. *The Collection Programming Language Reference Manual.* Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, October 1995. Available at `http://sdmc.iss.nus.sg/kleisli/psZ/cpl-defn.ps`.

[25] L. Wong. *The Kleisli/CPL Extensible Query Optimizer Programmer Guide.* Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, November 1995. Available at `http://sdmc.iss.nus.sg/kleisli/psZ/cplopt.ps`.