Turning Relational DBMS into Nested Relational DBMS

Jing Chen
GeneticXchange, Inc
chenjing@geneticxchange.com

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

ABSTRACT
We describe an interface to relational database management systems using SML and CML. This interface is used within the broad-scale bioinformatics data integration system called Kleisli. It extends the conventional flat relational data model supported by the query language SQL to a complex object data model. As a result, by going through this interface, a conventional flat relational data management system can be used transparently as a complex object store.

1. INTRODUCTION
Kleisli is a data integration system that has been used on biological data [21]. The primary query language provided by Kleisli is called CPL, which is a functional query language whose surface syntax is based on the comprehension syntax. Kleisli is itself implemented using CML [17], the concurrent version of the functional language SML[13]. Although Kleisli and CPL have previously been described [24, 23], the details of its interface to relational DBMS and other sources have never been published. In this paper, we describe the Kleisli's interface to relational DBMS such as Sybase [11], MySQL [26], and Oracle [10].

Most commercial interfaces to relational DBMS like ODBC [18], JDBC [14], Perl DBI [5], etc. are straightforward and just provide a means for sending SQL commands to the DBMS and retrieving results into internal memory structures that are essentially flat records. These interfaces also tend to be a suite of API to be called from some host programming languages and thus typically do not attempt any form of optimization. Also, some of these API do not shield the programmer from the differences between different DBMS and often require the programmer to know both SQL and his host programming language.

Kleisli's interface to relational DBMS differ from these typical interfaces. For example, even though a relational DBMS like MySQL handles only flat tables, the Kleisli interface endows the DBMS with the ability to handle deeply nested tables in a transparent manner. While some relational DBMS vendors have recently augmented their DBMS with object-relational capabilities [8, 10], the Kleisli's interface is considerably more natural and is founded on much firmer theoretical foundations [4]. Also, when the Kleisli's relational DBMS interface is accessed through CPL, Kleisli performs sophisticated optimizations to migrate operations from CPL to the DBMS to achieve better performance. In addition, CPL shields the programmer from differences in the SQL that different DBMS support. In fact, a CPL programmer only needs to know CPL and has no need to learn SQL.

We organize our presentation as follows. Section 2 uses the construction of a warehouse of protein sequence data to illustrate how a flat relational DBMS is effortlessly turned into a nested relational DBMS via CPL. Section 3 uses a query on a clinical database of Hepatitis B patients to illustrate how new powerful operators that cannot be expressed in SQL are smoothly grafted onto a relational DBMS via CPL. Both of these "demo" sections are presented using the syntax of CPL. Then in Section 4 we provide a detailed outline of the implementation of Kleisli's interface to relational DBMS that made the above possible. We also briefly discuss here Remy's technique for implementing constant-time polymorphic record projection [15] and the concept of "read ahead" laziness. Both of these ideas are important to the efficient functioning of the interface, We also discuss in Section 5 the four main optimization rules for queries on relational DBMS by migrating selection, projection,and joins from Kleisli to the DBMS. Both of these "implementation" sections are presented using actual codes from the Kleisli system in the syntax of SML/CML.

2. WAREHOUSE OF GENPEPT REPORTS
In the context of data integration, 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 is that the warehouse should model the data in a conceptually natural form. Although a relational database system is efficient for querying and easy to update, its native data model of flat tables often forces us to unnaturally and unnecessarily fragment our data in order to fit our data into the so-called third normal form [1].

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. The GenPept report is the format chosen by the US National Center for Biotechnology Information to present amino acid sequence information. Without getting into detail, it suffices to note here that the type of a GenPept report is the following complex type[1] discussed in [23]:

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

Kleisli provides several functions to access GenPept reports remotely from Entrez [19]: `aa-get-uid-general`, which retrieves unique identifiers of GenPept reports given a search string; `aa-get-seqfeat-general`, which retrieves GenPept reports given 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 2.1. *Create a warehouse of GenPept reports. Initialize it to reports on protein tyrosine phosphatases. 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 that each GenPept report is a highly nested complex object. There is therefore a "mismatch" between LONG (which is essentially a big uninterpreted string) and the complex structure of a GenPept report. This mismatch is resolved by the Kleisli interface to relational DBMS that automatically performs the appropriate encoding and decoding. Thus, as far as the Kleisli user is concerned,* `x.#detail` *has the type of GenPept report. So he can ask for the title of a report as straightforwardly as* `x.#detail.#title`.[2]

```
! 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};
```

□

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 2.2. *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.*[3]

```
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)};
```

[1] In the Kleisli system, the types are given by the grammar $t ::=$ `num` | `string` | `bool` | $\{t\}$ | $\{\!|t|\!\}$ | $[t]$ | $(l_1 : t_1, ..., l_n : t_n)$ | $\langle l_1 : t_1, ..., l_n : t_n\rangle$. Here `num`, `string`, and `bool` are the base types. The other types are constructors and build new types from existing types. The types $\{t\}$, $\{\!|y|\!\}$, and $[t]$ respectively construct set, bag, and list types from type t. The type $(l_1 : t_1, ..., l_n : t_n)$ constructs record types from types t_1, ..., t_n. The type $\langle l_1 : t_1, ..., l_n : t_n\rangle$ constructs variant types from types t_1, ..., t_n.

[2] The Kleisli user programs in a query language called CPL. The syntax of CPL is based on the comprehension syntax[3]. A typical comprehension in CPL syntax is $\{x * x | \backslash x \leftarrow S, odd(x)\}$, 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. See [24, 23] for more examples and explanation.

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

□

It would be nice to have 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 2.3. *A "memoized"* `aa-get-seqfeat-by-uid` *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* $f(v)$.

```
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);
```

Also, a "memoized" version of `aa-get-seqfeat-general`.

```
primitive my-aa-get-seqfeat-general == \s =>
 { x | \u <- aa-get-uid-general (s),
       \x <- my-aa-get-seqfeat-by-uid (u)};
```

□

In these examples, we encountered some of the functionalities provided by the Kleisli interface to Oracle, a popular relational DBMS. There is a `oracle-cplobj-add(#name: "db", ...)` construct that connects to an Oracle database and names the connection `db`. This "handle" to the connection in turns provides the functions `db-read`, `db-update`, `db-write`, `db-mktable`, `db-mkindex`, `db-insert`, `db-delete`, `db-droptable`, `db-dropindex`, and `db-query` for reading a table, updating a table, creating a table, defining a table, defining an index, inserting a record, deleting a record, deleting a table, deleting an index, and making an SQL query respectively. Also accesses via handles to different DBMS can be freely mixed.

It is worth pointing out two nice features of this Kleisli interface to relational DBMS. First, consider `db-update(#table: "genpept", #selector: (#uid: U), #replacement: (#detail: D))`. It corresponds to executing the SQL query `update genpept set uid = U where detail = D` on our Oracle database, except for the fact that `D` in this case is a complex object that could not normally be stuffed into Oracle unless some clever encoding could be done automatically. Second, consider `{x.#detail.#title | \x <- GenPept, x.#uid = 131470}`. It corresponds naively to `{x.#detail.#title | \x <- process "select * from genpept y where 1 = 1" using db-query, x.#uid = 131470}`, which causes the whole `genpept` table to be copied into Kleisli. However, Kleisli has an optimizer that is able to migrate operations from Kleisli into the external DBMS, and so the actual query that gets executed is the much more efficient `{x.#detail.#title | \x <- process "select detail = y.detail from genpept y where y.uid = 131470" using db-query}`, which instructs the DBMS to return precisely the record for 131470. We discuss the technical details later.

3. CLINICAL DATA QUERY

We discuss one other 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, ...}`.[4] 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 a big problem 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.

However, we can perform this query in Kleisli via a combinator for structural recursion [2]. The combinator is `list-sri`,

[4] 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.

which corresponds to the fold operation in functional programming languages.

EXAMPLE 3.1. *We can define a function* split *in terms of* list-sri *so that* split (d,c) S *returns a list* $[(x_1, Y_1)$, ..., $(x_n, Y_n)]$ *such that* Y_1, ..., Y_n *is* S *sorted chronologically;* x_1, ..., x_n *alternate between true and false; and* $c(y_j) = x_i$ *for each* $y_j \in Y_i$. *In the implementation below,* +] *inserts an element to the front of a list,* 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 let (\a, \b) == y.list-head in
    if a = x.check
    then (a, x +] b) +] 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,
                   \e == g.#2.list-rev.list-head.#date,
                   \b == g.#2.list-head.#date,
                   (e - b) >= 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 DBMS. Let us now proceed to the realm of SML/CML, the host implementation programming language environment of Kleisli, to see how this is achieved.

4. IMPLEMENTATION OF INTERFACE

Implementing an interface to relational DBMS for a programming language is a non-trivial task. Implementing an interface to a relational DBMS that endows it with nested relational capabilities is an even more challenging task. We divide the presentation of our solution in three subsections. Subsection 4.1 discusses the need to adopt a three-layer approach where user programs and the DBMS interface programs run on different threads autonomously and a server program in the middle coordinates their interactions. Subsection 4.2 discusses the implementation of complex objects that we used to represent the output from the DBMS. It also explains the concept of "read ahead" laziness that is central to the efficiency and function of the interface. Finally Subsection 4.3 gives the implementation of the interface proper.

4.1 Control

In designing our interface to relational DBMS, we must be cognizance of the following issues: (1) we may want to interface to more than one DBMS; (2) we may want to interface to other data sources; (3) the DBMS may live on a different machine; and (4) we may make more than one connection to the same DBMS. An implication of these issues is that the interaction model with the DBMS and other external data sources should be an asynchronous concurrent one.

This leads us to a three-layer design. The bottom layer consists of connections to DBMS and each connection should be placed on a different thread of execution. The top layer consists of the user's program threads that want to send requests to these connections. The middle layer consists of a server control to co-ordinate user requests and creation and destruction of connections.

Naturally, we needed concurrency. This requirement leads us to the choice of an implementation based on SML [13] and CML [16, 17]. In this subsection, we describe the middle layer server control module which is defined as a structure Server, whose main functions are given below.

```
structure Server = struct ...

fun ServerLoop SvNo PCh Dispatch =
 let (* Get next request *)
  val (ReplyCh, Req) = PChan.Accept PCh
  (* Create a control channel for connection
    to signify completion. *)
  val ReadyCh = CML.channel ()
 in (* Process it *)
  CML.spawn(fn() => Dispatch Req ReplyCh ReadyCh);
  (* Wait until connection is ready to do more. *)
  CML.accept ReadyCh;
  (* Repeat. *)
  ServerLoop SvNo PCh Dispatch end

fun SpawnServers SvNo PCh = map (fn (Dispatch, _) =>
 CML.spawn (fn() => ServerLoop SvNo PCh Dispatch))

fun ResetAllConnections SvNo =
let
 (* close old connectn *)
 val _ = CloseAllConnections SvNo
 (* open fresh ones *)
 val _ = OpenAllConnections SvNo
 (* create new request channel *)
 val PCh = PChan.MkChannel ()
 (* discard old channel *)
 val _ = ReqRepArray.update (ServerCh, SvNo, PCh)
 (* get connection's dispatch/shutdown function *)
 val DS = ConnectArray.sub (Connections, SvNo)
in
 (* Listen on all connections. *)
```

```
 SpawnServers SvNo PCh DS; () end

fun Access SvNo Priority Req = (
 case ConnectArray.sub (Connections, SvNo)
 of [] => Error.ReportFatal "no such connection,"
 | _ => let
   val ReqCh = ReqRepArray.sub (ServerCh, SvNo)
   val RepCh = CML.channel()
  in PChan.AsyncSend(ReqCh, Priority, (RepCh,Req));
     CML.accept RepCh end)

... end (* Server *)
```

In order to understand the codes above, we first describe some of the underlying data structures. First is an array `ServerCh` of prioritized channels for incoming requests; these channels are indexed by the unique identifier (usually denoted by `SvNo` in our codes) of the relational DBMS (or other kind of external data source) that the request is meant for. Prioritized channels are implemented in a module `PChan`. They are similar to ordinary CML channels, except that every message or object sent to a prioritized channel `PCh` is tagged with a priority. When a `PChan.Accept PCh` is executed, the message that has the highest priority in `PCh` is returned. Prioritized channels are very useful, for example, in the context of concurrent or parallel nested loops. If a user make requests to a DBMS in an outer loop and also in an inner loop, it is important that requests made from the inner loop be given higher priority. Otherwise, the requests from the inner loop will rapidly accumulate in a long queue and will potentially hog up resources. Second is an array `ServerLevel` of desired concurrency levels, again indexed by `SvNo`. Third is an array `Connections` of dispatch and shutdown functions, also indexed by `SvNo`. Thus, `ConnectArray.sub (Connections, SvNo)` is a list $[(D_1, S_1)$, ..., $(D_n, S_n)]$ of dispatch and shutdown functions that correspond to n active connections to the relational DBMS whose unique identifier is `SvNo`, where n is the desired concurrency level for this DBMS. Executing $S_i()$ shuts down this connection. Executing D_i r *replyCh readyCh* sends a request r to this connection; it also instructs the relational DBMS to send the result to a "reply" channel *replyCh*, as well as to signal completion by sending a notification to the "ready channel" *readyCh*. It is often desirable to have a few concurrent connections to the relational DBMS (or other external sources) to improve efficiency because the DBMS typically resides on a large physical server machine capable of handling a large number of requests simultaneously and also to exploit communication latency.

In order to make use of a relational DBMS, connections to that DBMS must be established. This is done by invoking the function `ResetAllConnections SvNo`, where `SvNo` is the unique identifier corresponding to that DBMS. As shown in the code fragment for `ResetAllConnections`, it opens connections to `SvNo`, creates a prioritised channel `PCh` to receive incoming requests for `SvNo`, and spawns new thread to run the service loop for this `SvNo` to listens and dispatches requests to `SvNo`. As shown in the code fragment for `ServerLoop`, in this new thread, the server control for an established connection of `SvNo`, whose dispatch function is `Dispatch`, awaits some incoming message on `PCh`. Such a message is pair `(RepCh, Req)`. `Req` is a complex object representing a request to be sent to `SvNo`. `RepCh` is the channel to be used by `Dispatch` for sending the answer or reply from `SvNo`. Next, a new "ready" channel `ReadyCh` is created. This channel is to be used by `Dispatch` to signal that it has completed dispatching `Req`. A new thread is then spawned to dispatch `Req`. The server control then waits for the ready signal on `ReadyCh`. Upon receiving the signal, we re-enter the loop to get ready for another request.

A user thread, on the other hand, calls `Access SvNo Priority Req` to make a request `Req` of priority `Priority` to `SvNo`. This function indexes the array `ServerCh` to obtain the prioritized request channel `ReqCh` of `SvNo`. It then creates a "reply" channel `RepChan`. It then sends `Req`, `RepCh` to `ReqCh` at the given priority. Finally, it awaits the response on `RepCh`. If there are several connections to `SvNo`, then `Req` can be dispatched by any of these connections that becomes free first, because the server control threads of these connections share the same request channel `ReqCh` as arranged by `ServerLoop`.

4.2 Complex Objects

Before we come to the implementation of our relational DBMS interface, it is also necessary for us to discuss in this subsection the representation of complex objects in Kleisli and also the concept of "read-ahead laziness."

The representation complex objects is shown below as a SML data type `CO`.

```
structure CO =struct ...

datatype CO
 = ...
 | String of string
 | EmptySet
 | SngSet of CO
 | UnionSet of CO * CO
 | lUnionSet of (unit -> CO) * (unit -> CO)
 | Record of CO Record.record

... end (* CO *)
```

Notice that sets is represented as a tree of unions of singletons using the constructors `EmptySet`, `SngSet`, `UnionSet`, and `lUnionSet`. The first three of these four set constructors have obvious meanings. The last one, `lUnionSet` is a little special. The `lUnionSet` construct allows a set to be built lazily. That is, whenever we have `lUnionSet` (f, g), it represents a set formed by the union of $f()$ and $g()$, where f and g are both expected to be functions that memoize[12] their results[5]

[5]We provide a function `Memoize` (f) in our system to convert any f of type `unit -> '1a` into a memoized function.

We have an important reason for choosing this "tree of unions of singletons" representation for sets. Kleisli is based on the notion of monads in the form of Kleisli triples [7]. Thus, its main operator is the homomorphism $\bigcup\{f(x) \mid x \in e\}$, which is the big union of $f(o_1)$, ..., $f(o_n)$ for the elements o_1, ..., o_n in e. This fact implies that the set union operation is expected to be the most frequently used operator in Kleisli. So it is desirable to make it as cheap as possible, and nothing is cheaper than making it a constructor!

Another item worth mentioning about `CO` is the representation of records. Traditionally, a record projection is compiled when all fields of the record are known in advance. Unfortunately, it is not possible to know all the fields of all the records that will be encountered in advance in our case. A simple way to deal with this situation is to simply represent record as a list of label-value pairs and thus make `type 'a Record.record = (Label * 'a) list`. However, in this case, the record projection becomes linear time, as it must be done by traversing the list to look for the pair having the right field name. Fortunately, Remy discovered an alternative method to define `Record.record` that has very low space overhead and enables constant-time record projection [22, 15].

Very briefly, Remy's method is as follows. We take advantage of the fact that when a record object is created, we know how many fields it has and what are the labels naming these fields. We assign to each label that we have seen a distinct number $\mathrm{A}(l)$. We do not have to know what is the type of each field of a record. Hence, we use the notation $[l_1, ..., l_n]$ to stand for the *abstract* type of a record which has exactly the fields named by l_1, ..., l_n. For each abstract type $[l_1, ..., l_n]$, we create a directory $\mathrm{D}_{[l_1,...,l_n]}$ as described below. The directory is an $m+1$ cell array of numbers. Here m is a 'magic' number such that $\mathrm{A}(l_1) \bmod m$, ..., $\mathrm{A}(l_n) \bmod m$ are distinct integers. Let cell 0 contains the number m. Let cell $1 + \mathrm{A}(l_i) \bmod m$ contains the number i, for $1 \leq i \leq n$. The contents of the remaining cells are not important. That is $\mathrm{D}_{[l_1,...,l_n]}[1 + \mathrm{A}(l_i) \bmod m] = i$, if $1 \leq i \leq n$. Each record object of abstract type $[l_1, ..., l_n]$ is stored as an array of $n + 1$ pointers. Let O be a typical record object whose abstract type is $[l_1, ..., l_n]$. We describe its contents below. In cell O[0], we store the pointer to the directory $\mathrm{D}_{[l_1,...,l_n]}$. In cells O[1], ...O[$n$], we store pointers to values of fields l_1, ..., l_n of the record object respectively. To compile record selection, say $r.l$, the following steps can be used. Let O be the record object that r has been compiled into. Then $\mathit{deref}(\mathrm{O}[\mathit{deref}(\mathrm{O}[0])[1 + \mathrm{A}(l) \bmod \mathit{deref}(\mathrm{O}[0])[0]]])$ fetches out the value of field l of r. This method is efficient in terms of both space and speed. In terms of speed, a record projection involves 4 array fetches, 2 pointer dereferencing, and 1 mod computation. This is the *constant* cost for access to any field of any record. In terms of space, there is 1 directory for each *abstract* type and one extra array slot for each record. A few cells get wasted if the corresponding m is not chosen wisely. This wastage is mitigated by the fact that different real records share the same directory. We performed experiments and verified that its speed overhead is at most eight times of the ideal scenario where all fields are known in advance. The overhead can be further reduced to at most three times the ideal scenario when sets are homogeneous[22].

From the definition of `CO`, it is clear that the "tree of unions of singletons" representation of sets allows some branches to be lazy and some branches to be non-lazy. For example, in `UnionSet` (`UnionSet` L, `lUnionSet` R)) L is non-lazy but R is lazy. Such mixing is useful because in the application environment of Kleisli—that of broad-scale bioinformatics data integration—the external data sources have different characteristics. For example, a small remote data source tends to be more efficiently accessed in an eager fashion. On the other hand, a large local file tends to be more efficiently accessed in a lazy fashion.

Besides allowing lazy branches to mix with non-lazy branches, there is another difference between our notion of laziness from the standard notion. Consider a function call $f(e)$, where f consumes the elements of the set e one by one. According to the usual notion of laziness as in Haskell [6], it is only when f demands the first element of e, then e is evaluated to something like `lUnionSet` (L_1, R_1), where L_1 is a singleton set and R_1 is an unevaluated closure. Similarly, it is only when f demands the second element of e, then R_1 is evaluated to something like `lUnionSet` (O_2, R_2), where O_2 is a singleton set and R_2 is an unevaluated closure. And so on. We can abstract from this description three properties of the usual notion of laziness. First, it is possible to return a partial result; for example, the partially evaluated e is returned to the subcomponent in f that demands it. Second, the evaluated part of a partial result is never re-evaluated. Third, the unevaluated part of the partial result remains unevaluated until demanded later; for example, R_1 is not evaluated if f does not demand the rest of e. Our notion of laziness possesses the first two properties. However, it sometimes *intentionally* ignores the third property. Thus in the function call $f(e)$, our system sometimes intentionally goes ahead to concurrently evaluate R_1, R_2, and so on even when f does not demand them, and it also makes L_1, L_2, and so on available to f without waiting for R_1, R_2, and so on to finish.

There are two main reasons for this "read ahead" form of laziness. The first reason is to enable parallelisation of access to certain powerful external data sources. For example, $f(e)$ might be a CPL program, `{ g(x) | \x <- e}`, where $g(x)$ tries to insert or obtain record from a remote relational DBMS using an element x of e. A relational DBMS typically runs on a powerful server machine and can usually handles many thousand transactions per minute. Since it is remote, there is a significant amount of latency in transmitting a request to it and in receiving a reply from it. It would be a wasted opportunity if we have to wait until the first $g(x)$ is finished running (ie. the first x is sent to the DBMS and a reply is received) before we start the second

$g(x)$. It would make a lot of sense to run as many $g(x)$ in parallel as the DBMS can sustain. This means the evaluation of e should not wait for g to ask for an x. It also means that we should not wait for all the x's to be available before running $g(x)$. "Read ahead" laziness is the natural solution to this situation. The second reason is to enable proper access to replies that an external server delivers sequentially in a text stream. It is often most convenient to access an external server by a pair of text streams, one for sending requests and one for receiving replies. Text streams are serial in nature. That is, it is not possible to read the reply to the second request without first reading the reply to the first request. In a fully lazy system such as Haskell, an insufficiently skilled programmer might attempt to read the second reply without first finish reading the first reply and thus hangs his system. As with most SQL programmers of commercial DBMS, most CPL programmers are not that skillful, which is why they needed a high-level system like Kleisli. So our system must prevent this kind of problem from occurring. The solution is to automatically use "read ahead" laziness on access to text streams.

As CO are the representation of complex objects, it is thus the representation of the output from relational DBMS. We expect the user would typically want to perform extensive manipulations on them that are not supported by the DBMS, as shown in the "window" example from the previous section. So we also provide a large collection of useful functions on CO objects, such as structural recursion, set intersection, and so on.

4.3 Interface

We are now ready to implement our interface to relational DBMS. We show only a fragment of the interface to Oracle. Other relational DBMS such as Sybase and MySQL can be interfaced to similarly. The interface consists of the main function OracleGenDirty.

```
structure ORACLE_CPL_OBJ = struct ...

fun OracleGenDirty Host Port Sid User Passwd () =
let
 (* Connects to the Oracle system *)
 val (RepStr, ReqStr) = CIO.execute (OraCodes,
  [Host, Port, Sid, User, Passwd])

 (* Function to write one record *)
 fun WriteToOracle F X =
 let
  val (Ls, Vs) = CORecord.KmRaw X
  val Vs = map CPL2Oracle Vs
  val Ls = fold (fn (l, a) => if a = ""
   then StringUtil.suffix (l,1)
   else (StringUtil.suffix (l,1))^", "^a) Ls ""
  val Vs = fold (fn (l, a) => if a = ""
   then l else l ^ ", " ^ a) Vs ""
  val Q = "INSERT INTO "^F^" ("^Ls^") " ^
         "VALUES ("^Vs^");\n"
 in
  (* Send query to database *)
  CIO.output (ReqStr, Q);
  (* Get reply. *)
  COI.ReadFromStream RepStr end

 (* Function to process insert requests. *)
 fun DispatchInsert Req ReplyCh ReadyCh =
 let
  val t = COString.Km (CORecord.Proj LTable Req)
  val r = CORecord.Proj Lreplacement Req
 in
  CML.send (ReplyCh, WriteToOracle t r);
  CML.send (ReadyCh, ()) end

 (* Function to query DB directly. *)
 fun DispatchQuery Req ReplyCh ReadyCh =
 let
   val Q = COString.Km Req
   val ReplyCh' = CML.channel ()
   val ReadyCh' = CML.channel ()
   fun Cvt CO.EmptySet = CO.EmptySet
   | Cvt (CO.SngSet x) = (case x
      of CO.Record _ => COSet.Singleton (
          CORecord.Map Oracle2CPL x)
     | _ => COSet.Singleton x)
   | Cvt (CO.UnionSet(x,y)) = CO.UnionSet (Cvt x, Cvt y)
   | Cvt(CO.lUnionSet(x,y))=COSet.lUnion(Cvt o x)(Cvt o y)
   | Cvt _ = Error.ReportFatal ...
 in
  (* Send query to database *)
  CIO.output (ReqStr, Q ^ ";\n\n");
  (* Send reply and signal ready. *)
  CML.spawn (fn() => COI.ReadFromStreamLazy
    ReplyCh' ReadyCh' RepStr);
  CML.send (ReplyCh, Cvt(CML.accept ReplyCh'));
  CML.send (ReadyCh, CML.accept ReadyCh') end

...

in ([DispatchInsert,DispatchQuery, ...],ShutDown) end

... end (* structure ORACLE_CPL_OBJ *)
```

The call OracleGenDirty Host Port Sid User Passwd () logs on to an Oracle relational DBMS. This is done by performing a system call to our JDBC driver to Oracle OraCodes. This system call produces two text streams RepStr and ReqStr. The ReqStr stream is for us to send SQL commands, via the JDBC driver OraCodes, to the DBMS for execution. The RepStr stream is for us to read the response, in Kleisli's exchange format [25], via the JDBC driver OraCodes, from the DBMS. In the body of OracleGenDirty, these two streams are accessed by various "dispatch" functions specialized for dealing with specific SQL commands for update. These "dispatch" functions and a function to "shutdown" the connection are returned by OracleGenDirty. Thus after invoking OracleGenDirty, the user can use these resultant functions to send requests to the DBMS via the resulting connection.

We discuss here the dispatch function DispatchInsert for inserting a new record into a table for illustration. Other dispatch functions such as DispatchMkIndex, DispatchDelete, etc. are omitted for brievity. As mentioned earlier, a dispatch function is intended to be run on a separate thread; it takes in a request Req (which is a CO object) to be processed, a "reply" channel ReplyCh to be used for sending the result (which is also a `CO` object), and a "ready" channel `ReadyCh` to be used for signaling the completion of the request to the server control thread. In the case of `DispatchInsert`, the request `Req` is expected to be a complex object[6] `(#table:` F, `#replacement:` X`)`, where X is the record to be inserted into the table F. `DispatchInsert` extracts F and X using the appropriate record projections. It then applies `WriteToOracle` to caused the DBMS to insert X into F. After that, it signals ready. The function `WriteToOracle` extracts from X the fields `Ls` and values `Vs` of the record to be inserted, forms an SQL insert command using these fields and values, and then dispatch this SQL insert command to the DBMS for execution. An important thing that `WriteToOracle` must take care of is that `Vs` may contain other complex objects and so it must encodes them into strings first. The encoding is achieved by mapping the encoding function `CPL2Oracle` over each value in `Vs`. A variety of encoding methods can be used, for example, by escaping quotation marks and unprintable characters. However, a smart encoding method like `CPL2Oracle` distinguishes atomic types such as numbers and strings from complex types such as records and sets to reduce computational and space overheads of encoding; in particular, `CPL2Oracle` is simply the identity function on atomic types!

As another example, we discuss `DispatchQuery`. As with all dispatch functions, it is intended to be run on a separate thread. It is for sending a string representing an SQL query to the DBMS for execution. So the `Req` is extracted as a string Q. A fresh "reply channel" `ReplyCh'` and a fresh "ready" channel is created by `DispatchQuery` for each new query. It prints (ie. sends) Q to the text stream `ReqStr` so that the DBMS can receive and executes Q. It then spawns a new process to wait for the reply on the text stream `RepStr` from the DBMS and to lazily parse the result using `COI.ReadFromStreamLazy` in a "read ahead" manner. `COI.ReadFromStreamLazy` is used for two reasons: (1) the result can be a very large set and it is often better to return a large set lazily so that processing of its elements can begin without waiting for the whole set to be ready; and (2) it is necessary to read ahead because so that we can clear the text stream `RepStr` for the next result, as explained in the previous subsection. The result that `COI.ReadFromStreamLazy` delivers to the reply channel `ReplyCh'` is potentially encoded, as it preactically comes direct from the DBMS. So a decoding function `Cvt` must be applied to it before delivering the final result to the reply channel `ReplyCh`. The decoding function `Cvt` is a homomorphism on the internal representation of complex objects that maps `Oracle2CPL`, the inverse of `CPL2Oracle`, to top level records in the result.

Note that the dispatch functions returned by `OracleGenDirty` should not be used directly. Instead, they must be registered under different `SvNo` using a function (not shown) in the server control module `Server`, and then used indirectly via a call to access function `Access` in `Server`. This is to ensure that the necessary coordination between the multiple threads of execution is achieved. Of course, if you are using strictly CPL to query relational DBMS, this is automatically taken care of!

[6]Where there is little chance of confusion, we use the CPL syntax here and elsewhere to more succintly reveal the structure of the complex object, rather than the official SML syntax in terms of constructors from the `CO` structure.

5. OPTIMIZATION

In this section, we present the implementation in SML of the main rules used in Kleisli for optimizing queries on relational DBMS. We first briefly revisit the core of the Kleisli optimizer, previously presented in [24, 23]. Then we go into details of migrating operations to relational DBMS from Kleisli.

5.1 Optimizer Core

The type `SYN` of SML objects that represent queries in Kleisli is given below. The types `VAR` and `SVR` stand respectively for variable names and server connection names. We explain only the less obvious constructors. The constructor `ExtSet`(e_1, x, e_2) is the natural transformation on monads[4, 20] and thus represents an expression that evaluates to the set $\bigcup\{e_1 \mid x \in e_2\}$. The constructor `Read`$(s, p, e)$ corresponds the CPL expression for passing a request e to a server connection s, `process e using s`, where p is the priority given to this request. The constructor `Binary`(f, e_1, e_2) is for caching static objects; it is a means for the optimizer to insert codes for doing dynamic optimization.

```
datatype SYN = ...
| EmptySet
| SngSet of SYN
| UnionSet of SYN * SYN
| ExtSet of SYN * VAR * SYN
| IfThenElse of SYN * SYN * SYN
| Read of SVR * real * SYN
| Variable VAR
| Binary (CO * CO -> CO) * SYN * SYN
```

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

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

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; and so on, together with their multi-pass versions. Using higher-order functions, all of these rule application strategies are decomposed into a "traversal" component that is common to all strategies and a very simple "control" component that is special for each strategy.

In particular, the "traversal" component is a higher-order SML function `Decompose: (SYN -> SYN) -> SYN -> SYN` that is shared by all strategies. It 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 : `RULEDB -> SYN -> SYN` is a function that 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. We show just one below.

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

```
fun TopDownOnce RDB Qry =
 let
  fun Pass Subqry = Decompose Pass (
   case Pick RDB SubQry
   of SOME BetterSubQry=> BetterSubQry
   | NONE => SubQry)
 in Pass Qry end
```

□

5.2 Relational Optimization

Relational DBMS are themselves equipped with the ability to perform sophisticated transformations expressed in SQL. A good optimizer (and a good interface) 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 5.2. *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.*

```
fun FullyProjected x N =
 let val (Count, Projs) = (ref 0, ref [])
  (* function to count projections *)
  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
  (* rule base containing just FindProjs *)
  val RDB = {
   DoTrace = ref true,
   Trace = ref (fn _=> fn _=> fn _=> ())
   Rules = ref [("", FindProjs)]}
 in TopDownOnce RDB N;
    if length (!Projs) = !Count
    then SOME (!Projs) else NONE
 end
```

Recall that the CPL construct `process` M `using` S *is represented in SML 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))) =
 (* test if S connects to a SQL server *)
 if Annotations.IsSQL S
 (* test if x is always in a projection *)
 then case FullyProjected x N
  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 $\texttt{x.\#}l_1 = c$, then we should push as many of these tests to the database as possible.

EXAMPLE 5.3. *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* $\texttt{if } (\bigwedge Pve) \wedge \neg(\bigvee Nve) \texttt{ then } N' \texttt{ else } \{\} = N$. *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.* `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 and so cannot be migrated. 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))) =
 (* test if S is a relational db *)
 if Annotations.IsSQL S then
  let
   (* can't migrate tests on BLOB-like fields *)
   val Forbidden = SQLDict.ExtractLONG S((!SQL.Mk) M)
   val H = IntHashTable.mkTable(1,Error.Goofed "Oops!")
   val _ = IntHashTable.insert H (x, Forbidden)
   val (Pve, Nve, N') = FlattenTests (CheckPushTest H) N
   (* convert list to tower of if-then-else *)
   val Pve = CvtTest S Pve
   val Nve = CvtTest S Nve
  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 $\texttt{x.\#}l_1 = \texttt{y.\#}l_2$, 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 5.4. *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))) =
 (* test if S is a SQL server *)
 if Annotations.IsSQL S
 (* test if x is always in a projection *)
 then case FullyProjected x N
  of SOME ProjO =>
   (* ProjO are projections on the outer relation *)
   let
    (* can't migrate tests on BLOB-like fields *)
    val Forbidden = SQLDict.ExtractLONG S ((!SQL.Mk) M)
    val H = IntHashTable.mkTable(10,Error.Goofed "Oops!")
    val _ = IntHashTable.insert H (x, Forbidden)
    val (PO, NO, N') = FlattenTests (CheckPushTest H) 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
       projs 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 H 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 5.5. *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))) =
 (* test if S is a relational database *)
 if Annotations.IsSQL S then
  let
   (* Vars are free variables in N *)
   val Vars = SyntaxTools.FV N
   (* can't migrate BLOB-like fields *)
   val Forbidden =  SQLDict.ExtractLONG S ((!SQL.Mk) M)
   val H = IntHashTable.mkTable(1, Error.Goofed "Oops!")
   val _ = IntHashTable.insert H (x, Forbidden)
   fun Ins y = IntHashTable.insert H (y, fn _ => false)
   val _=IntSet.app(fn y=>if y = x then() else Ins y)Vars
   (* 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 Vars' = IntSet.difference(Vars, IntSet.singleton x)
   (* Pve, Nve are tests that can be migrated dynamically.
      N' is what's left. *)
   val (Pve, Nve, N') = FlattenTests (fn N =>
    (CheckPushTest H N) andalso
    (IntSet.member(SyntaxTools.FV N, x))) N
  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}`. However, 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 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.

6. CONCLUDING REMARKS

We have disected the implementation of Kleisli's interface to relational DBMS. We have also seen how it turns these DBMS from flat to nested relational. While we mostly use this interface through the high-level query language CPL, it is should be evident that the interface can just as easily be accessed as an API from CML.

In comparison to the better known interfaces to relational DBMS such as ODBC, JDBC, etc., our interface has several advantages. It effectively "upgrades" all underlying DBMS to nested relational DBMS, giving them the power to model and store complex objects. It also seemlessly integrates operations and optimizations of CPL with that of all underlying DBMS, migrating operations that could be performed more efficiently by these DBMS to them.

There is another effort in interfacing functional programming languages to relational DBMS known to us. It is the HaskellDB [9] project. The higher-order typed mechanism of Haskell is put to good use to embed the abstract syntax of SQL into Haskell as a set of primitive constructors and comprehension monads. Then a special function is used to dynamically generate the corresponding SQL queries for execution on a relational DBMS via ODBC. This interface is remarkably natural and the Haskell programmer never has to leave the familiarity of Haskell. Its focus is on getting the types correct. It is not known to be able to turn relational DBMS into nested relational DBMS the way we do. It is not known to be able to perform the relational optimizations we have described. It is not concurrent. Another place for HaskellDB to improve upon is that its embedding could not express a query like $\{e_1 \mid \backslash x$ `<-` R, $\backslash y$ `<-` S, $e_2\}$ where R and S are tables from two different databases. The HaskellDB is at an early stage, it might eventually be provided with these features.

Although the "domain embedding" approach of HaskellDB does not provide as seamless an integration with relational DBMS as achieved by CPL, it has the great advantage of not having to build a full-fledge query language like we did with CPL. We believe that it will be fruitful to combine the two approaches by layering a similar kind of domain embedding on top of the various structures underlying our relational DBMS interface.

7. REFERENCES

[1] S. Abiteboul et al. *Foundations of Databases.* Addison-Wesley, 1995.

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

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

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

[5] A. Descartes and T. Bunce. *Programming the Perl DBI.* O'Reilly, 2000.

[6] J. H. Fasel et al. The functional programming language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[7] H. Kleisli. Every standard construction is induced by a pair of adjoint functors. *Proc. Amer. Math. Soc.*, 16:544–546, 1965.

[8] V. Krisnamurthy et al. Bringing object-relational technology to the mainstream. *SIGMOD Record*, 28(2):513–514, 1999.

[9] D. Leijen and E. Meijer. Domain specific embedded compilers. *Proc. 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.

[10] K. Loney and G. Koch. *Oracle8i: The Complete Reference.* McGraw-Hill, 2000.

[11] D. McGovern and C.J. Date. *A Guide to SYBASE and SQL Server.* Addison-Wesley, 1992.

[12] D. Michie. Memo functions and machine learning. *Nature*, 268:19–22, 1968.

[13] R. Milner et al. *The Definition of Standard ML.* MIT Press, 1990.

[14] G. Reese. *Database Programming with JDBC and Java.* O'Reilly, 2000.

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

[16] J. Reppy. CML: A higher-order concurrent language. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[17] J. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[18] R. E. Sanders. *ODBC 3.5 Developer's Guide.* McGraw-Hill, 1998.

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

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

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

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

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

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

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

[26] R. J. Yarger et al. *MySQL & mSQL.* O'Reilly, 1999.