THE Kleisli/CPL EXTENSIBLE QUERY OPTIMIZER
Programmer Guide

Limsoon Wong

Institute of Systems Science
Heng Mui Keng Terrace, Singapore 0511
`email: limsoon@iss.nus.sg`

16 April 1996

Version C Release 16 April 1996

Contents

1 Introduction

1.1 Introduction

Kleisli is a system for querying heterogeneous data sources that are distributed over a wide area network and are not necessarily relational [5]. It is being used to provide on-demand integration of a number of databases in the Human Genome Project [13]. It is implemented in Concurrent ML [21]. On top of Kleisli is a high-level query language called CPL, for Collection Programming Language. Both CPL and much of the internal structure of Kleisli is based on the notion of monads [7, 6].

In this report, the focus is on the optimizer of the Kleisli/CPL system. As mentioned earlier, the system is being applied to query heterogeneous biological data sources. This application environment has the following characteristics.

- The data sources are remote and thus have to be accessed via a wide-area network. Therefore, the amount of data transfer and the amount of network idle time can have an important impact on performance. This is a situation not normally encountered by relational optimizers, for most of them are used only for local queries.

- The data sources are heterogeneous both in terms of data models and systems and these are being added to our system over time. Therefore, the extensibility of the optimizer has an impact on our ability to exploit the capabilities of new data sources. This is also not found in traditional relational optimizer since their data model is very homogeneous and their systems are usually inextensible.

- The data sources have complex structures. For example, one of our most important genetic database is GenBank [17]. A sequence record in this database typically has over 12 levels of nesting and 150 subobjects. Therefore, the optimizer has deal with queries that are considerably more complex than those supported by relational databases.

- The schemas and statistics of the data sources can be nonexistence or inaccessible. Therefore, the optimizer has to function in the absence of such information. This is in contrast to all optimizers of relational databases, which require such information to produce good query plans.

- Queries in Kleisli/CPL can use operations that are not available on any of the source systems themselves. Therefore, the optimizer must take special care in deciding where to dispatch various operations. Furthermore, some of these operations have to be imported from other existing systems into Kleisli/CPL. Therefore, the extensibility of the optimizer also has an impact on our ability to exploit the capabilities of imported operators. This is again in contrast to most relational optimizer that are implemented for closed systems.

The Kleisli/CPL optimizer consists of an extensible number of phases. Each phase consists of a rule base and a rule application strategy. The system currently has six basic srategies and several rule bases for optimization such as pipelining, join-migration, and so on. More rule bases and strategies can be added to exploit the capabilities of new data sources. In fact, the system provides very high-level support for developing new rule application strategy and new optimization rules.

1.2 Organization

Section 2. An example query joining two databases is given to introduce some features of Kleisli/CPL and optimization ideas. In particular, the idea of migrating joins to data sources is used to improve overall performance, the idea of lazy evaluation is used to reduce response time, and the idea of parallelism is used to reduce network idle time.

Section 3. The representation of queries in our monad-based framework is discussed. In particular, the advantage of this framework in simplifying the programming of optimization rules is explained.

Section 4. The overall architecture of the optimizer is described. In particular, the representations of rules and strategies are given and the basic technique for generating rule application strategies in the optimizer is explained.

Section 5. A variety of optimization rules and their actual implementation codes is presented. Rules for pipelining, parallelism, code motion, join-migration, and so on are included in this discussion. Further examples are also given in this section.

Section 6. A brief comparison our optimizer to some other extensible rule-based optimizers is given. Then some enhancements being planned are described.

1.3 Notations

In this report, I will be writing pieces of CPL codes and pieces of SML codes. (Remember, CPL is implemented using various library modules of Kleisli, which are in turn implemented on top of Concurrent ML and uses SML [15] as the host programming language.) To avoid confusion, I will use different fonts for these languages. For CPL codes, I will use `type-writer` font. For SML codes, I will use *italic* font in general, but I will use san-serif font to highlight the specific SML constructs that are used for representing CPL queries.

1.4 Acknowledgement

I would like to thank Susan Davidson, David Maier, and Jonathan Smith for pushing me in this direction. I also thank Jonathan Crabtree and Kyle Hart for the query examples.

2 Motivating Example

In this section, we use an example query that joins data from two relational databases to illustrate some features of the Kleisli/CPL system and some optimizations performed by our optimizer.

Note that Kleisli/CPL supports complex objects where sets, bags, and lists can be arbitrarily nested [5]. Furthermore, many data sources in our Human Genome Project environment are non-relational. The use of the relational example below is to simplify explanation. Everything in this paper applies to non-relational queries in our system.

2.1 Original Query

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

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

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

Let us use the query above to briefly explain the syntax of CPL, the query language of our system. More details can be obtained from the *CPL Reference Manual* [28], or in a slightly different form from [6, 5].

Access to remote data sources is expressed by the `process` *REQUEST* `using` *SERVER* construct. The query above uses two servers: `gdb` and `sortez`. The first one provides access to GDB at John Hopkins in Baltimore. The second provides access to Sortez at Penn in Philadelphia. Both of these are Sybase relational databases. The `process-using` construct works by sending the *REQUEST* to the specified *SERVER* and then parsing the corresponding reply into a complex object. Kleisli/CPL supports multiple connections to all of its data servers. For the discussion in this section, we use one connection for GDB and two for Sortez.

The most important query construct in CPL is the comprehension construct. For sets, this construct usually has the form `{ E | P1 <- E1, ..., Pn <- En, C1, ..., Cm}`. E, E_i, and C_j can be arbitrary expressions. However, E_i must evaluate to sets and C_j must evaluate to Booleans. Each P_i is a pattern used for matching elements in the corresponding set E_i. Thus the construct can be roughly interpreted in psuedo SQL as

> select E
> from E_1 $P1$, ..., E_n P_n
> where C_1 and ... and C_m

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

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

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

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

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

It is obvious that such a query will take a long time to produce any output, even if we have an unconjested high-capacity T3 link between Singapore and the United States, let alone the conjested T1 link at the moment of writing. It is clearly unacceptable. In the rest of this section, we describe some optimization ideas. The codes for implementing these ideas and others will be given in a later section.

2.2 Maximal Subquery Migration

The obvious optimization is to migrate operations to data sources whenever possible. The rationale for this optimization is that certain statistics and indices are available to the native optimizers at some of these data sources. Therefore, they can be expected to perform the operations more efficiently. In fact, we should migrate subqueries that are as large as possible [18] to execute remotely at the data sources. The effect of migrating maximal subqueries is to ensure that the native optimizers have as much context as possible and to give them as much work as possible.

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

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

The equivalent query after the migration of the three-way join needed to compute `loci22` to GDB is shown above. As can be seen clearly, `loci22'` now sends just one SQL request to GDB to obtain the desired three-way join. So executing `combine'` requires merely one access to GDB and 20 accesses to Sortez.

Furthermore, each of the 20 accesses to Sortez in `combine'` is a great deal cheaper than that in `combine`. The reason is that `combine` brings in the entire `gb_head_accs` table each time, whereas `combine'` brings in only a few columns of that table (because the projections on these columns have been migrated) and only a few rows of that table (because the selection on the `pastaccn` column has been migrated).

This is a big improvement for we can now complete the query in about 42 seconds. However, we can improve further. In this version of the query, the first record is not displayed until the entire result table is computed. Hence, its response time is also around 42 seconds. If we are joining more records than the first 20, we will have to wait a rather long time to see anything.

2.3 Lazy Evaluation

To improve response time, we should start the output process as soon as some rows in the resulting table become available. This can be accomplished by lazy evaluation. For the concept and theory of lazy evaluation, please consult functional programming texts such as [9].

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

The idea of lazy evaluation is implemented in the query above. It is exactly the same as `combine'`, except the presence of a `l`-tag in the comprehension construct used in `combine''`. This tag indicates that the result of the comprehension is to be formed lazily. That is, as soon as some elements of the set become available, the system should return them, without waiting for the rest of the elements in the set. The rest of the elements are returned on demand when they become available. CPL takes care of coordination automatically.

Thus while `combine''` requires the same number of accesses to GDB and Sortez as `combine'`, it starts displaying the first row of its result after one access to GDB and one access to Sortez. The response time is now about 7 seconds, greatly reduced from the 42 seconds of `combine'`. However, it still needs about 42 seconds to complete the query. We can improve on this 42 seconds total time by observing that access to Sortez in this version is essentially sequential.

2.4 Parallel Evaluation

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

```
primitive combine''' ==
   {l (#gdb: x,
       #sortez: {(a,b,c,d,e)
                 | (#locus: \a, #accn: \b, #title: \c, #len: \d, #taxname: \e)
                       <- process "select locus = g.locus, accn = g.accn,
                                          len = g.len title = g.title,
                                          taxname = g.taxname
```

```
                              from gb_head_accs g
                              where g.pastaccn = " ^ sql-string(r)}
                        using sortez)
   | \x & (#genbank_ref: \r, ...) <<- loci22'};
```

The idea of using parallelism to reduce waiting is implemented in the query above. It is exactly the same as `combine''`, except the replacement of `<-` by `<<-`. In CPL, P `<-` E means matching the pattern P over elements of the set E in a sequential fashion. On the other hand, P `<<-` E means matching the pattern P over elements of the set E in parallel. Thus `combine'''` takes about 10 second to display the first row and finishes the last row in about 19 seconds, using two connections to the `sortez` server in parallel.

In introducing such parallelism, we must be careful of two things. First, the remote server may only be able to handle a limited number of requests at a time, say five. In this case, we should send five requests at a time to avoid overwhelming the remote server. Fortunately, data servers in our application are usually running on top of commercial systems like Sybase which practice admission control. In addition, Kleisli/CPL provides mechanism to adjust its level of concurrency with respect to different servers.

Secondly, each concurrent thread requires resources such as memory to be allocated if their output are not consumed quickly enough. This is tougher to deal with than the first problem. Kleisli/CPL relies on a priority-based scheme to schedule its external data requests. In Section 5, an optimization rule for assigning priority and its impact on a larger query will be given.

2.5 Optimization Results at a Glance

We tabulate below the typical execution time with respect to the number of `loci22` entries. All results are obtained by running our queries on a SPARC 20 in Singapore. As mentioned before, the data sources are located in the east coast of the United States in Philadelphia (for Sortez) and Baltimore (for GDB). All accesses rely on a T1 link between the two countries. Note that for `combine'''` we use two connections to Sortez.

		Number of Entries					
		20	50	80	110	140	170
Total Time (sec)	`combine'`	42	53	77	109	133	179
	`combine''`	26	63	91	142	142	159
	`combine'''`	19	37	49	64	80	100
	C Program	25	51	84	94	120	146

		Number of Entries					
		20	50	80	110	140	170
Response Time (sec)	`combine'`	42	53	77	108	132	178
	`combine''`	7	11	8	7	9	13
	`combine'''`	10	9	8	7	8	14

We have included the running time of a C implementation of the same query using the sequential nested loop algorithm similar to `combine''`. We note that our parallel version of the query, `combine'''`, is consistently faster then the C version. All this has been achieved using CPL without the pains of complicated programming. Later in the paper, we will describe the CPL optimizer that will automatically perform all of the above optimizations.

Note that if write permission is granted on Sortez, the alternative optimization of semi-join [25] can be used to achieve potentially greater improvement than the parallel version of `combine'''` above. However, the parallel idea of `combine'''` is a more general form of optimization than the semi-join idea. In particular, the semi-join idea works well only if the data source is a powerful system such as a full-fledged relational database — Unfortunately, this kind of data sources is in the minority in the current application environment of the Human Genome Project.

3 Query Representation

Before we can proceed to describe the optimizer, we need to describe the representation of CPL queries in SML, the host programming language of Kleisli. While CPL uses the comprehension syntax [6] as its surface syntax, its internal representation is based on the nested relational calculus derived from the categorical notion of a monad [7, 27].

3.1 Query Representation

The actual implementation of the representation of CPL queries in SML is given below. We omit bags, lists, variants, and a few other constructs as they are not needed for the purpose of this report. Consult the *Kleisli Query System Reference Manual* [30] for full detail. We use the terms "query", "query representation", and "query plan" interchangeably in this report.

```
datatype SYN
  = Bool of bool
  | IfThenElse of SYN * SYN * SYN
  | EmptySet
  | SngSet of SYN
  | UnionSet of SYN * SYN
  | ExtSet of SYN * varname * SYN
  | IExtSet of SYN * varname * SYN
  | IIExtSet of SYN * varname * SYN
  | IIIExtSet of SYN * varname * SYN
  | Record of remyrecord
  | Proj of label * SYN
  | Read of server * priority * SYN
  | Variable of varname
  | Fun of varname * SYN
  | Apply of SYN * SYN
  | Primitive of primname
  ⋮
  | Nullary of compobj
  | Unary of (compobj → compobj) * SYN
  | Binary of (compobj * compobj → compobj) * SYN * SYN
```

As can be seen from the above, CPL queries are represented by objects of type SYN in SML. The constructors Bool and IfThenElse represent Boolean constants and `if-then-else` construct in CPL. The constructors Record and Proj represent the formation and the field selection constructs for records in CPL. Note that the details of a record such as its field labels and values at those fields are packed away in a *remyrecord*, which is a representation based on an idea of Remy [20] as described in [29]. It is unnecessary to know the details to program the optimizer.

The Variable, Fun, and Apply constructors correspond to referencing a variable, forming a function, and applying a function. The Primitive constructor is for referencing an external or imported primitive. The Read constructor is used to represent the `process-using` construct in CPL. Thus the expression `process` R `using` S `at priority` P in CPL is represented as $\mathsf{Read}(S, P, R)$ internally in our system.

The Nullary, Unary, and Binary constructors are used to represent partially evaluated queries. The evaluated part is either a fully evaluated complex object value, indicated by *compobj*; a compiled unary function together with its unevaluated argument, indicated respectively by $compobj \rightarrow compobj$ and SYN; or a compiled binary function together with its two unevaluated arguments, indicated respectively by $compobj * compobj \rightarrow compobj$ and $\mathsf{SYN} * \mathsf{SYN}$. Note that the unary and binary functions can be anything, such as ones that call the optimizer at runtime based on the value of the arguments passed to them at runtime. Thus these constructors can be used to implement a more dynamic form of optimization rules. An example will be given in Subsubsection 5.5.2.

We use the constructors EmptySet, SngSet, UnionSet to represent the CPL constructs for forming emptyset, singleton set, and set union. The ExtSet, lExtSet, llExtSet, and lllExtSet constructors indirectly represent the comprehension construct in CPL. The actual meaning of $\mathsf{ExtSet}(N, x, M)$ is the set $f(o_1) \cup \cdots \cup f(o_n)$, where $f(x) = N$ and $\{o_1, \ldots, o_n\} = M$. This representation is based on viewing sets as a monad; see [27, 7]. We discuss its virtue in Subsection 3.2. lExtSet, llExtSet, and lllExtSet have exactly the same denotational meaning as ExtSet, but with slightly different operational behaviours. Note that the UnionSet constructor also comes in the same four flavours as ExtSet. We omit them in this report.

We expand on their differences here, since the optimizations in Section 2 use rules involving them; see Section 5. ExtSet corresponds to sequential iteration in an eager manner — it is compiled into a loop that iterates over the elements of M sequentially and it returns its result all at once. lExtSet corresponds to sequential iteration in a lazy manner — it builds its result a few elements at a time on demand rather than all at once. llExtSet corresponds to parallel iteration in an eager manner — it iterates over elements o_i of M in parallel, waits for all instances of $N[o_i/x]$ to complete, and then returns their union in one shot. lllExtSet corresponds to parallel iteration in a lazy manner — it iterates over elements o_i of M in parallel and returns which ever instance $N[o_i/x]$ that is ready first, and returns other instances on demand when they become ready later.

3.2 Wadler Identities

Let us now explain the correspondence between the concrete comprehension syntax of CPL and the SYN-representation described in the last subsection. Essentially, comprehensions are recursively translated into the SYN-representation using the identities of Wadler [26], with minor extensions to deal with laziness and parallelism, as shown in the table below.

CPL	SYN Object
$\{N \mid \}$	$\mathsf{SngSet}(N)$
$\{N \mid \backslash x$ <- $M, \ldots\}$	$\mathsf{ExtSet}(\{N \mid \ \ldots\}, x, M)$
$\{N \mid M, \ldots\}$	$\mathsf{IfThenElse}(M, \{N \mid \ \ldots\}, \mathsf{EmptySet})$
$\{l\ N \mid \}$	$\mathsf{SngSet}(N)$
$\{l\ N \mid \backslash x$ <- $M, \ldots\}$	$\mathsf{lExtSet}(\{l\ N \mid \ \ldots\}, x, M)$
$\{l\ N \mid M, \ldots\}$	$\mathsf{IfThenElse}(M, \{l\ N \mid \ \ldots\}, \mathsf{EmptySet})$
$\{N \mid \backslash x$ <<- $M, \ldots\}$	$\mathsf{llExtSet}(\{N \mid \ \ldots\}, x, M)$
$\{l\ N \mid \backslash x$ <<- $M, \ldots\}$	$\mathsf{lllExtSet}(\{l\ N \mid \ \ldots\}, x, M)$

The concrete comprehension syntax of CPL is extremely user-friendly. However, user-friendly is not always optimizer-friendly. Since optimizer has to manipulate representation of queries, it is important that we choose a representation that is as friendly to the optimizer as possible. Otherwise, we would be forced to write unnecessarily complicated codes for the optimizer and it would be messy to implement even the simplest optimization rules.

We originally planned to build an optimizer generator based on some ideas from [12, 8]. However, as can be seen later in Section 5, the translation of CPL into the monad-based representation makes it possible for us to develop our optimizer, rules, and strategies in so simple a manner that an optimizer generator becomes unnecessary. In fact, optimization rules can some time be written in our system in a simpler manner than in an optimizer generator!

4 Optimizer Architecture

We describe the architecture of our optimizer in this section. We also show here how easy it is to generate different strategies for applying rules.

4.1 Overall Architecture

The Kleisli/CPL optimizer consists of an extensible number of phases. Each phase consists of a rule base and a rule application strategy. The system currently has six basic srategies and several rule bases for optimization such as pipelining, join-migration, and so on. More rule bases and strategies can be added to exploit the capabilities of new data sources. In fact, the system provides very high-level support for developing new rule application strategy and new optimization rules.

The composition of the optimizer is depicted in the diagram below. The main components of the optimizer are the strategies (such as BottomUpMany and MaxOnce) and the rule bases. Currently, two of our rule bases make use of meta information. The strategies are all generated from the Decomposer. More complex strategies based on mixing these and other strategies can also be generated on the fly; examples will be given later in Subsection 4.4.

```
                +--------------+
Unoptimized---->| Multiphase   |---->Optimized
Query           |  Optimizer   |     Query
                +--------------+
                       |
     +------+----------+-------+---------+------+------+--------+
     |      |          |       |         |      |      |        |
+------+ +------+ +--------+ +--------+ +----+ +----+ +------+ +-----+
|Bottom| |Bottom| |TopDown| |TopDown| |Max | |Max | |Entire| |Rule |
|UpMany| |UpOnce| |Many   | |Once   | |Many| |Once| |Once  | |Bases|
+------+ +------+ +--------+ +--------+ +----+ +----+ +------+ +-----+
     |       |         |         |       |      |               |
     +-------+-----+---+---------+-------+------+               |
                   |                                            |
            +--------------+                            +-----------+
            |  Decomposer  |                            | Meta Info |
            +--------------+                            +-----------+
```

The main phases of the optimizer are described in Subsection 4.2. The representation of optimization rules is described in Subsection 4.3 and examples of rules in Section 5. We now briefly describe the available strategies. Their implementation will given later after the Decomposer is described in Subsection 4.4.

To describe the behaviour of the rewrite strategies, it is convenient to think of the representation of a query as a tree that grows downwards in the usual computer science fashion, with nodes tagged by constructors such as ExtSet, IfThenElse, etc. For example, if the query is IfThenElse(E_1, E_2, E_3), we can think of it as a tree having three branches. The root of the tree is a node tagged by the constructor IfThenElse. The three branches are three trees corresponding to E_1, E_2, and E_3.

The BottomUpOnce strategy traverses a query representation from leaves up to the root. It rewrites each node of the tree at most once. The BottomUpMany strategy simply repeats the BottomUpOnce strategy many times until no rule can be applied.

The TopDownOnce strategy traverses a query representation from the root down to the leaves. It rewrites each node of the tree at most once. The TopDownMany strategy simply repeats the TopDownOnce strategy many times until no rule can be applied.

The MaxOnce strategy also traverses a query representation from the root down to the leaves. It also rewrites each node at most once. However, in contrast to TopDownOnce, after rewriting a node, it stops considering the subtrees at that node. In other words, it finds and rewrites only the maximal components of the query. The MaxMany strategy simply applies MaxOnce many times until no rule can be applied.

The EntireOnce strategy does not traverse a query representation. It simply applies rewrite rules to the root of the query. Its main purpose is to offer the programmer an escape to invoke an alternative optimizer — it can be used to apply a single rule whose meaning is "use the alternative optimizer XYZ."

4.2 Main Optimization Phases

The standard set up of the Kleisli/CPL system uses the phases below in its optimizer. Examples of their associated rules and rationale for their associated rule application strategies are given in Section 5.

Phase 1. This is the de-annotation phase and is carried out using the BottomUpOnce rule application strategy. The purpose of this phase is to strip off various control tags such as the `l`-tag used for expressing laziness from the query representation. The standard set up assumes that it is the optimizer's responsibility to decide on laziness and parallelism. Hence in the first phase of optimization, it discards all such control tags.

Phase 2. This is the monad optimization phase and is carried out using the TopDownMany rule application strategy. The purpose of this phase is to reduce the query representation into a certain normal form according to a set of rules derived from the axioms of monads. Optimizations such as constant folding, vertical loop fusion, filter promotion, etc. are performed as a result of these rules. Most importantly, this set of rules is a superset of that used in [27] to prove the conservative extension property of nested relational calculus. Thus the resulting normal form is very suitable for subsequent analysis of what subqueries can be migrated to be performed at the data sources.

Phases 3 This phase augments the query with projection information. It inspects the parts of the query that involve external relational databases. It tries to replace the "*" in as many "select * from table where conditions" in these parts by the actual fields. This is done by looking up meta information, if it is available. This step enables more operations to be migrated to these external relational databases later on. This phase is applied using the BottomUpOnce strategy.

Phase 4. This phase uses exactly the same set of rules and the same rule application strategy as Phase 2. Its purpose is to tidy up the query representation resulting from the previous augmentation phase.

Phases 5-8. These phases are responsible for migrating as many operations to relational servers at the data sources as possible. They are carried out using the BottomUpOnce (for Phases 5 and 7) and the TopDownOnce (for Phases 6 and 8) strategies. Phases 5-8 migrate maximal subqueries involving one relational database: union operations are migrated in Phase 5, selections are migrated in Phase 6, joins are migrated in Phase 7, and projections are migrated in Phase 8.

Phase 9. This phase uses exactly the same set of rules and the same rule application strategy as Phase 2. Its purpose is to tidy up the query representation resulting from the migration phase.

Phase 10. The projection information introduced in Phase 3 can now be discarded.

Phase 11-12. These two phases consider joins between two different external relation databases. Phase 11 uses meta information to decide which database should be used as the inner relation of the join. Phase 12 discovers the join predicates between them and perform transformations that have the effect of migrating these predicates dynamically. Phase 11 uses the BottomUpMany strategy, while Phase 12 uses the TopDownOnce strategy.

The selection on the `pastaccn` column in the `combine'` query in Section 2 is an illustration of how Phase 12 rewrites `combine` to achieve the effect of dynamic migration of join predicates between two databases.

Phase 13. This phase uses exactly the same set of rules and the same rule application strategy as Phase 2. Its purpose is to tidy up the query representation resulting from the dynamic migration phase.

Phase 14. This is the code-motion phase and is carried out using the MaxMany rule application strategy. The migration of invariant codes out of loops [2] and the sharing of common subexpressions are achieved in this phase. The use of the MaxMany strategy ensures that only maximal code fragments are migrated or shared.

Phase 15. This is the prioritization phase and is carried out using the TopDownOnce rule application strategy. In this phase, priorities are assigned to all data server accesses in the query representation. The overall effect of this optimization is to ensure that external data requests that lead to an overall reduction in resource utilization are dispatched first.

Phase 16. This is the operator specialization phase and it is carried out using the BottomUpOnce rule application strategy. In general, polymorphic operation such as the polymorphic equality test `x = y` is more expensive to execute than monomorphic one. So we have a set of simple rules to do specialization of comparison operation.

Phase 17. This is not really an optimization phase. It happens to be convenient to use the optimizer to perform certain initializations such as clearing buffers and caches. It is carried out using the EntireOnce rule application strategy.

Phase 18. This phase replaces repeated Remy projections on polymorphic but homogeneous records by direct jumps. It is carried out using the EntireOnce strategy. This is a low-level optimization exploiting certain properties of the runtime representation of records in Kleisli. The basic idea and the impact of this optimization are explained in [29].

Phases 19-21. These are the re-annotation phases. They are carried out using the BottomUpOnce rule application strategy. In essence, the rules in these phases decide which parts of the query are to be executed lazily, eagerly, and/or in parallel.

4.3 Rule Representation

In this subsection we briefly discuss the representation of rules, strategies, and the structure of rule bases in our optimizer.

4.3.1 Rewrite Rules

A rewrite rule R is represented in our system as a ML function of type

$$\mathsf{SYN} \rightarrow \mathsf{SYN}\ \mathit{option}$$

If a rule R is applicable to a query plan Q to produce an improved plan Q', then the outcome $R(Q)$ is *SOME*(Q'). Otherwise $R(Q)$ is *NONE*.

4.3.2 Rule Bases

A rule base RDB is represented in our system as a ML record of type

{ *DoTrace: bool ref,*
Trace: (rulename → SYN → SYN → *unit) ref,*
*Rules: (rulename * (*SYN → SYN *option)) list ref*}

The *Rules* field of RDB stores the list of rules in RDB together with their names. The *Trace* field of RDB stores the function f that is to be used for tracing the usage of the rules in RDB. The *DoTrace* field of RDB stores a flag to indicate whether tracing is to be done.

If tracing is indicated, then whenever a rule of name N in RDB is applied successfully to transform a query Q to Q', the trace function will be invoked as $f\ N\ Q\ Q'$ to record a trace. Normally, this simply means a message like "Q is rewritten to Q' using the rule N" will be printed. However, the trace function f is allowed to carry out considerably more complicated activities. Section 5 has several examples that exploit trace functions to achieve sophisticated transformation in a simple way.

Let us make clear the above description by showing the *Pick* function that is used for picking a rule in RDB to rewrite the $Query$. It tries the rules in RDB one by one, until one of them succeeds. It is implemented by the following piece of ML codes in our system.

```
fun Pick RDB Query =
  let val DoTrace = !(#DoTrace RDB)
      val Trace = !(#Trace RDB)
      fun Pick' [] = NONE
      | Pick' ((Name, Rule)::OtherRules) =
          case Rule Query
          of SOME ImprovedQuery ⇒ (
              if DoTrace then Trace Name Query ImprovedQuery else ();
              SOME ImprovedQuery)
          | NONE ⇒ Pick' OtherRules
  in Pick' (#Rules RDB)
  end
```

4.3.3 Strategy Representation

A rule application strategy S is a ML function of type

```
{ DoTrace: bool ref,
  Trace: (rulename → SYN → SYN → unit) ref,
  Rules: (rulename * (SYN → SYN option)) list ref}
→ SYN → SYN
```

That is, the rule application strategy S takes in a rule base RDB to produce a function that transforms a query representation into an improved one. In other words, S embodies a strategy of using the rules in RDB to transform query representations.

4.4 Strategy Generation

In this subsection we demonstrate a simple and powerful technique for generating rule application strategies. This technique is based on the use of a Decomposer.

4.4.1 Decomposer

The Decomposer is a higher-order function of type

```
(SYN → SYN) → SYN → SYN
```

The Decomposer accepts a rewrite function f and a query representation Q. Then it applies f to all the 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 is therefore very straightforward.

```
fun Decompose f (SngSet N) = SngSet(f N)
|    Decompose f (UnionSet (N, M)) = UnionSet (f N, f M)
|    Decompose f (ExtSet (N, x, M)) = ExtSet (f N, x, f M)
⋮
```

Above are a few lines of ML codes taken from our actual implementation of the Decomposer in the Kleisli/CPL system. Please consult [15] for the syntax of ML. In the remaining subsections, this simple function is used to construct many useful optimizer strategies.

4.4.2 Implementing MaxOnce

Recall that MaxOnce is the strategy that tries to apply rules to maximal subtrees. So it should start trying the rules on the root of the query representation. If no rule can be applied, it should move down one level and re-try. But as soon as a rule can be applied, it should stop at that level. Its implementation in our system is given by the following remarkably short and clear piece of ML codes.

```
fun MaxOnce RDB Query =
    case Pick RDB Query
        (* A rule exists to rewrite Query to ImprovedQuery. Apply and stop. *)
    of SOME ImprovedQuery ⇒ ImprovedQuery
        (* No rule applies. Go down one level. *)
    | NONE ⇒ Decompose (MaxOnce RDB) Query
```

Since this is the first rule application strategy we have presented, let us give an unusual but useful example to illustrate its use. The example is unusual because it has nothing to do with applying a rewrite rule to optimize queries. Nevertheless, it is extremely useful in its own right: Given a predicate and a query representation, test whether some subcomponent of the query representation satisfies the predicate. It also shows an interesting use of "trace" function to perform tasks other than "tracing" the application rules.

```
fun Thereis Pred Query =
    let (* Initialize existence flag to false. *)
        val Exist = ref false
        (* A rule base to set flag to true on Pred satisfication. *)
        val RDB = {
            DoTrace = ref true,
            Trace = ref (fn _ ⇒ fn _ ⇒ fn _ ⇒ Exist := true),
            Rules = ref[("", fn SubQuery ⇒ if Pred Subquery then SOME Subquery else NONE)]}
    in  (* Use MaxOnce to try Pred on components of Query. *)
        MaxOnce RDB Query;
        (* Return the value of the existence flag *)
        !Exist
    end
```

Recall from Subsection 4.3.2 that if a rule from a rule base is applied and if *DoTrace* of that rule base is set to *true*, then the corresponding function stored at the *Trace* field of the rule base will be invoked. In the implementation of *Thereis* above, this feature is used to set the *Exist* flag.

4.4.3 Implementing TopDownOnce

Recall that TopDownOnce is a rule application strategy that traverses a query representation from the root down to the leaves, It tries to improve each node of the tree at most once as it progresses to the leaves. It can also be implemented in an extremely simple piece of ML codes using the Decomposer. Here it is:

```
fun TopDownOnce RDB Query =
  let fun Pass Subquery =
    let (* First rewrite the root. *)
      val ImprovedSubquery =
        case Pick RDB Subquery
        of SOME ImprovedSubquery ⇒ ImprovedSubquery
        | NONE ⇒ Subquery
    in (* Then go down the levels. *)
      Decompose Pass ImprovedSubquery
    end
  in Pass Query
  end
```

4.4.4 Implementing BottomUpOnce

Recall that BottomUpOnce is the rule application strategy that traverses a query representation from the leaves up to the root. During the traversal, it tries to improve each node once. Its implementation is similar to that of TopDownOnce, with the call sequence switched in the obvious way.

```
fun BottomUpOnce RDB Query =
  let fun Pass Subquery =
    let (* First go down the levels. *)
      val ImprovedSubquery = Decompose Pass Subquery
    in (* Then rewrite the root. *)
      case Pick RDB ImprovedSubquery
      of SOME FurtherImprovedSubquery ⇒ FurtherImprovedSubquery
      | NONE ⇒ ImprovedSubquery
    end
  in Pass Query
  end
```

4.4.5 Implementing Other Strategies

Other rule application strategies can also be implemented in a very straightforward way. For example, the MaxMany strategy, which simply repeats the MaxOnce strategy many times, can be implemented as follows:

```
fun MaxMany RDB Query =
  let (* Again is false if no rule applies last time. *)
      val Again = ref false
      (* Implement a version of MaxOnce that sets the Again flag if it succeeds. *)
      fun RewriteMax Subquery = (
          case Pick RDB Subquery
          of SOME ImprovedSubquery ⇒ (Again := true; ImprovedSubquery)
          | NONE ⇒ Decompose RewriteMax Subquery)
      (* Rewrite using MaxOnce strategy. *)
      val ImprovedQuery = RewriteMax Query
  in  if !Again
      then  (* Some rule applies last time. So do again. *)
            MaxMany RDB ImprovedQuery
      else  (* No change. So do no more. *)
            ImprovedQuery
  end
```

Essentially, we just apply MaxOnce and set the *Again* flag if MaxOnce produces an improved query. Then we test the flag. If it has been set, we reset it and repeat the whole procedure. The BottomUpMany and TopDownMany strategies are implemented using exactly the same idea. The default strategy called EntireOnce has an even simpler implementation.

As can be seen from the preceding examples, many different rule application strategies can be generated by factoring through the Decomposer. I think it is even more remarkable that the Decomposer is itself a simple function — one that simply applies a given strategy to immediate subtrees of a given query plan.

It should also be pointed out that in all our examples of rule application strategy above, there is no explicit codes that involve the actual definition and structure of query plan. Everything that needs to know the details of query representation is fully encapsulated within the Decomposer. By supplying a different Decomposer, the rule application strategies developed as shown above can immediately be applied to a completely new query language. We have thus found a good modularization of query optimizers.

5 Optimization Rules

We show how easy it is to implement a wide range of interesting optimization rules in this section. All of them are used by our optimizer.

5.1 Rules for Monad

As mentioned in the beginning of this report, our Kleisli/CPL system is designed around a framework inspired by monads [16]. Some of the benefits of this approach became apparent from our previous work. It gives us the very user-friendly comprehension syntax [6, 26, 23, etc.] It gives us a very uniform treatment of collection types [7]. And it gives us a very clean basis for studying expressive power of query languages [27, 14, 22, etc.]

In this subsection, two additional benefits of this approach become apparent. The first benefit is the representation of queries presented in Section 3. It is almost trivial to implement many optimization rules using this representation of queries. The second benefit is the equational theory of monads. Its axioms give us several powerful and general optimization rules. We illustrate these two points using the rules for vertical loop fusion [11] and filter promotion [4]. Other rules derived from the axioms of monads can be found in [7, 27], their implementation is just as simple.

5.1.1 Vertical Loop Fusion

Vertical loop fusion[11] is closely related to the concept of pipelines and quantifier elimination. It is an optimization that combines two loops into one to reduce the size of intermediate data. It is applicable when the first loop is a producer and the second loop is a consumer.

This optimization idea is expressed in comprehension syntax as a rule that rewrites an expression $\{e \mid \ldots,$ $\backslash x$ `<-` $\{e' \mid \ldots\}, \ldots\}$ to $\{e[e'/x] \mid \ldots, \ldots, \ldots[e'/x]\}$. The effect is the elimination of the intermediate set $\{e' \mid \ldots\}$. While this rule is very easy to comprehend by a human being, it is extremely messy to implement directly because a computer must be explicitly told how to deal with the highly informal "...".

Fortunately, we do not have to worry about this problem because our internal representation of queries does not use comprehension syntax. As revealed in Section 3, we use the more regular form that corresponds to the Kleisli presentation of monads. Thus vertical loop fusion is expressed as a rule that rewrites an expression $\mathsf{ExtSet}(e_1, x, \mathsf{ExtSet}(e_2, y, e_3))$ to $\mathsf{ExtSet}(\mathsf{ExtSet}(e_1, x, e_2), y, e_3)$. This rule is implemented directly in our system using the following trivial piece of ML codes:

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

5.1.2 Filter Promotion

Filter promotion [4] corresponds to the well-known concept of moving test predicates closer to generators [24]. It is closely related to the idea of pushing selections toward base tables in relational optimization [25].

The optimization idea can be expressed in comprehension syntax as a rule that transform an expression $\{e \mid \ldots, \backslash x$ `<-` $e', \ldots, p(x), \ldots\}$ to $\{e \mid \ldots, \backslash x$ `<-` $e', p(x), \ldots, \ldots\}$. While a human being can easily see that this rule simply moves the test $p(x)$ to right behind the generator $\backslash x$ `<-`e', it is rather messy to implement directly because a computer has to be told what those "..." mean.

Fortunately, we again do not need to worry about such a problem. Filter promotion is expressed using our representation of queries as a rule that rewrites $\mathsf{ExtSet}(\mathsf{IfThenElse}(e_1, e_2, e_3), x, e_4)$ to $\mathsf{IfThenElse}(e_1, \mathsf{ExtSet}(e_2, x, e_4), \mathsf{ExtSet}(e_3, x, e_4))$, provided x is not free in e_1. Here is the corresponding piece of straightforward ML codes:

```
fun FilterPromotion (ExtSet(IfThenElse(E1, E2, E3), x, E4)) =
    if SyntaxTools.IsFree x E1
    then NONE
    else SOME(IfThenElse(E1, ExtSet(E2, x, E4), ExtSet(E3, x, E4)))
  | FilterPromotion _ = NONE
```

To see that we have indeed implemented filter promotion, consider the expression $\{e_1 \mid \backslash x$`<-`$e_2, \backslash y$`<-`$e_3, p(x)\}$. The representation of this query is $\mathsf{ExtSet}(\mathsf{ExtSet}(\mathsf{IfThenElse}(p(x), \mathsf{SngSet}(e_1), \mathsf{EmptySet}), y, e_3), x, e_2)$. Applying the transformation above, we get $\mathsf{ExtSet}(\mathsf{IfThenElse}(p(x), \mathsf{ExtSet}(\mathsf{SngSet}(e_1), y, e_3), \mathsf{ExtSet}(\mathsf{EmptySet}, y, e_3)))$. Since $\mathsf{ExtSet}(\mathsf{EmptySet}, y, e_3)$ can be rewritten to $\mathsf{EmptySet}$ using another rule, we get $\mathsf{ExtSet}(\mathsf{IfThenElse}(p(x), \mathsf{ExtSet}(\mathsf{SngSet}(e_1), y, e_3), \mathsf{EmptySet}), x, e_2)$. This result is of course the representation of the query $\{e_1 \mid \backslash x$`<-`$e_2, p(x), \backslash y$`<-`$e_3\}$.

5.2 Rules for Constant Folding

In this subsection, we implement an optimization idea called constant folding [2]. It is a slightly more complicated kind of rewrite rules that requires the optimizer to interact with the query evaluation engine of the Kleisli/CPL system.

5.2.1 Constant Folding

Constant folding [2] is the detection of subqueries that can be cheaply evaluated during query optimization, as opposed to during normal query evaluation. These are generally subqueries that do not depend on external data.

The basic idea in implementing constant folding is to first detect subqueries that are worth folding. Then invoke the Kleisli/CPL query evaluation engine on these subqueries. Then translate the results back into query representation. Here is the SML codes:

```
fun FoldConstant Query =
    if Foldable Query
    then SOME (DlofConstant (Interpretor.RunSilent Query))
    else NONE
```

In the above, *Foldable* is a predicate that tests whether *Query* requires access to external data, raises explicit exceptions, has free variables, or contains side effects. It is implemented using the existence test given in Subsubsection 4.4.2 as follow:

```
val Foldable = not o Thereis (
    fn (Apply (Primitive P, _)) => Annotations.HasSideEffect P
    | (Error _) => true
    | (Read _) => true
    | (Variable _) => true
    | _ => false)
```

Also, *RunSilent* is a function that invokes the query evaluation engine on *Query*. The result is produced as a fully evaluated complex object. So the function *DlofConstant* is used to convert it back into the query representation format SYN. Conversion is simple because SYN comes with the Nullary constructor; see Section 3.

5.2.2 Operand Commutation

To make the constant folding rule more effective, the commutativity and associativity of operators have to be considered. Otherwise, it would not be possible to rewrite $3 + x + 5$ to $8 + x$ for example.

A rule that simply rewrite $f(e_1, e_2)$ to $f(e_2, e_1)$ when f is commutative cannot be added to our system. The reason is that we assume that the rules in any single rule base form a strongly normalizing rewrite system. Such a rule for commutativity would violate strong normalization, since it can easily lead to an infinite non-terminating rewrite sequence.

However, it is possible to implement a restricted form of commutativity rule that is sufficient for the purpose of increasing the chance of constant folding. The basic idea of this restricted rule is to swap the left and right argument of a commutative operator only if the right argument is foldable but the left is not. The asymmetry in this rule thus assures that it will not be applied infinitely. Below is the implementation. The rule for associativity is implemented in a similar manner.

```
fun Commute (Apply (Primitive P, Record R)) =
    if not (Annotations.HasSideEffect P) andalso (Annotations.IsCommutative P)
    then let val (E1, E2) = Record.KmPair R
         in if (Foldable E2) andalso not (Foldable E1)
            then SOME (Apply (Primitive P, Record (Record.MkPair E2 E1)))
            else NONE
         end
```

```
        else NONE
    | Commute _ = NONE
```

5.3 Rules for Context-Sensitivity

In this subsection we present an interesting class of rules that requires the use of multiple rule application strategies. The scope of the rules in the previous subsection is over the entire query plan. In contrast, each of the rules in this section has two parts.

The inner part is "context sensitive" and its scope is limited to certain component of the query plan. The outer part scopes over the entire query plan to identify contexts where the inner part can be applied. The two parts of the rule can be applied using completely different strategies.

5.3.1 If-Then-Else Absorption

An example is the rule that rewrites `if` e_1 `then` ... e_1 ... `else` e_3 to `if` e_1 `then` ... `true` ... `else` e_3. The inner part of this rule rewrites e_1 to `true`. The outer part of this rule identifies the context and scope of the inner part of this rule: limited to the `then`-branch.

The above example is very intuitive to a human being. In the `then`-branch of a conditional, all subexpressions that are identical to the test predicate of the conditional must eventually evaluate to `true`. However, such a rule is not so straightforward to express to a machine. The informal "..." are again in the way.

Fortunately, rules of this kind are straightforward to implement in our system. The above example is expressed by the *AborbThen* rule below. The rule has three clauses. The first clause says that the rule should not be applied to an IfThenElse whose test predicate is already a Boolean constant, because it would lead to non-termination otherwise. The second clause says that the rule should be applied to all other forms of IfThenElse. The third clause says that the rule is not applicable in any other situation.

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

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

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

5.3.2 Read Prioritization

Consider a nested loop where the same server S is accessed in both the outer loop and the inner loop. As illustrated in Section 2, data parallel iteration over these loops lead to significant improvement. Thus at any one time, a number of requests would be pending dispatch to S.

The order in which these requests are dispatched has an effect on performance. Suppose all requests from the outer loop are dispatched before those from the inner loop. Then evaluation becomes essentially breadth-first. This has two problems. Firstly, the response time is slow because no iteration of the outer loop can complete until the last request to S from the outer loop is dispatched. Secondly, a large amount of resources such as memory is needed because no iteration of the outer loop can release its memory until the last request to S from the outer loop is dispatched.

On the other hand, suppose we favour requests from the inner loop over those from the outer loop. Then evaluation starts to take on a depth-first characteristic. Initially some requests to S from the outer loop are dispatched. Results from these initial requests cause a few iterations from the inner loop to begin. This leads to some requests to S from the inner loop. Now there is a mix of requests to S from both inner and outer loops. Those from the inner are dispatched. Once the replies come back, the corresponding iterations of the outer loop can complete. Consequently, response is faster and resources are released faster.

This optimization idea is captured by a rule that increases the priority of Read accesses inside a loop. The implementation below is a rule that uniformly adds 1 to all Read that is inside a ExtSet-loop. See how we use *ContextSensitive* to restrict the *IncPriority* function to the body *E1* of the loop and use the TopDownOnce strategy to apply it to all the Read within scope.

```
fun Prioritize (ExtSet (E1, x, E2)) =
    let fun IncPriority(Read (S, P, R)) = SOME (Read (S, P + 1.0, R))
        | IncPriority _ = NONE
    in case SimpleOptimizer.ContextSensitive IncPriority SimpleOptimizer.TopDownOnce E1
        of SOME E1' => SOME (ExtSet(E1', x, E2))
        | NONE => NONE
```

end
| *Prioritize* _ = *NONE*

The above rule is used in the prioritization phase of our optimizer. This phase itself uses the TopDownOnce strategy. Thus this rule would be applied at every node of the query plan from root to leaves. The overall effect is that the priority of each Read in inner loops is increased corresponding to the depth at which it is found.

We give a real example to illustrate the impact of this rule. Below is a CPL query that accesses GenBank via two connections to the Entrez server in Washington D.C and GDB via one connection to a Sybase server in Baltimore. The query itself is issued from Philadelphia. This query extracts the protein translation of genes on Chromosome 22.

```
{l (#gbname: gbname, #length: l, #accnum:a,
    #gdblocus:gl, #definition: gbtitle, #translation:t) |
      \g & (#acc: \a, #gdblocus: \gl) <<-
          process
            "select distinct acc = gbref.genbank_ref, gdblocus = x.locus_symbol
             from locus_released x, object_genbank_eref gbref
             where x.locus_type_code <> 'M' and x.locus_id
                in (select locus_id
                    from locus_cyto_location lcl
                    where lcl.loc_cyto_chrom_num = '22'
                    and x.locus_type_code in ('G'))
                    and x.locus_id = gbref.object_id"
          using gdb,                                    ! outermost loop.
      <#seq: (#id: \ids, #descr: \descrs,
              #inst: (#length: \l, ...), ...)> <<-
                  process (
                     #db: "na", #path :"Seq-entry",
                     #select: "accession " ^ a, #args: "")
                  using entrez,                         ! middle loop.
      <#genbank: (#name: \gbname,...)> <- ids,
      <#gi: \gid> <- ids,
      <#title: \gbtitle> <- descrs,
      (#seq_data: \t,...) <-
         {i | <#seq: (#inst:\i, ...)> <-
                process (
                   #db:"na", #path: "Seq-entry", #select:"",
                   #args: ("-k aa  -g " ^ num-stringify(gid)))
                using entrez }                           ! innermost loop.
}
```

Entrez is accessed from both the middle loop and the innermost loop. If Entrez is accessed with equal priority from both loops, this query takes an average of 460 seconds to response with some output and an average of 480 seconds to complete. If Entrez is accessed with higher priority from the innermost loop, this query takes an average of 320 seconds to response and an average of 440 seconds to complete.

5.3.3 Invariant Code Motion

Code motion is the concept of moving codes that do not change value out of a loop, thus saving repeated computations [2]. In our case, a special form of invariant code motion corresponds to a rule that rewrite { ... e ... | \$x_1$`<-`$e_1$, ..., \$x_n$`<-`$e_n$} to `let` \$y$ `==` e `in` { ... y ... | \$x_1$`<-`$e_1$, ..., \$x_n$`<-`$e_n$}, where e is an expensive expression that does not depend on x_1, ..., x_n.

Again such a rule is easy to comprehend by a human being, while the "..." makes it hard to give a straightforward description to a machine. Fortunately, the ability to invoke dynamically created rules context-sensitively in our system comes to the rescue. The ML codes below implement a code motion rule that is more general that the special case mentioned in the previous paragraph.

```
fun CodeMotion (ExtSet(E1, x, E2)) =
  let (* The set of variables staying constant. *)
      val Context = SyntaxTools.FV (ExtSet(E1, x, E2))
      (* Store invariant subexpressions here. *)
      val Moveable = ref []
      (* Move invariant expressions here. *)
      val Y = FreshVar ()
      val P = ref 0
      (* Context-sensitive rule for moving invariants. *)
      fun Move E = if WorthMoving E andalso IntSet.isSubset (SyntaxTools.FV E, Context)
         then (inc P;
               Moveable := E :: (!Moveable);
               SOME (Proj ( Label.Mk ("#" ^ (makestring (!P))), Variable Y)))
         else NONE
  in (* Use MaxOnce to move maximal invariant codes. *)
      case SimpleOptimizer.ContextSensitive Move SimpleOptimizer.MaxOnce E1
      of SOME E1' => SOME (GEN.MkLet Y
         (Record (Record.MkTuple (rev (!Moveable))))
         (ExtSet (E1', x, E2)))
      | NONE => NONE
  end
| CodeMotion _ = NONE
```

It rewrites $\mathsf{ExtSet}(e_1, x, e_2)$ to a query plan corresponding to `let` \$y$ `==` $(a_1, ..., a_n)$ `in` $\mathsf{ExtSet}(e_1', x, e_2)$. The a_i's are all the expensive subexpressions of e_1 that do not depend on x and any loop variables in e_1; that is, the free variables in each a_i are a subset of the free variables of $\mathsf{ExtSet}(e_1, x, e_2)$. And e_1' is obtained form e_1 by replacing each a_i by $\mathsf{Proj}(i, y)$.

The rewriting of e_1 to e_1' corresponds to the inner part of a context-sensitive rule. It is implemented by the *Move* function. Since we should pull expressions that are as large as possible out of the loop, we apply *Move* using the MaxOnce strategy.

Incidentally, the optimization rule for sharing expensive common subexpressions can be implemented in a very similar way.

5.4 Rules for Laziness and Parallelism

As illustrated in Section 2, the use of laziness and parallelism can lead to much better performance. The Kleisli/CPL system allows different parts of a query to be evaluated in different modes. In this subsection, we discuss the simple rules we use to assign evaluation modes to different parts of a query. We first made everything lazy. Then we identify those operators that need to examine their input repeatedly and make their input eager. Finally, we identify those loops that access external sources and make them parallel. These three stages correspond to Phases 19, 20, and 21 mentioned in Subsection 4.2.

5.4.1 Lazy Evaluation

Recall that to specify that a set is to be constructed lazily, it is only necessary to insert a `l`-tag into a comprehension, rewriting a CPL query $\{e \mid ... \}$ to $\{l\ e \mid ... \}$. The corresponding rule in our SYN-representation of query is a function that rewrites ExtSet to lExtSet and llExtSet to lllExtSet. Here is the straightforward ML codes implementing this rule in our system.

```
fun Lazy (ExtSet (E1, x, E2)) = SOME (lExtSet (E1, x, E2))
| Lazy (llExtSet (E1, x, E2)) = SOME (lllExtSet (E1, x, E2))
| Lazy _ = NONE
```

5.4.2 Eager Evaluation

While lazy evaluation generally reduce response time, as well as peak memory usage, it comes with some overhead. Thus it should be avoided when it is not needed [1]. In general, laziness is not needed if a primitive has to completely examine its input argument before producing any output. In Kleisli/CPL such primitives satisfy the *IsEager* test. Thus,

```
fun Eager (Apply (Primitive P, E)) =
    if Annotations.IsEager P
    then case SimpleOptimizer.ContextSensitive Deannotate SimpleOptimizer.TopDownOnce E
        of SOME E' ⇒ SOME (Apply (Primitive P, E'))
        | NONE ⇒ NONE
    else NONE
| Eager _ = NONE
```

As shown above, when we encounter a function application involving an eager primitive, we simply remove all `l`-tags from its input argument. The function to strip `l`-tags is shown below.

```
fun Deannotate (lExtSet (E1, x, E2)) = SOME (ExtSet (E1, x, E2))
| Deannotate (llExtSet (E1, x, E2)) = SOME (ExtSet (E1, x, E2))
| Deannotate (lllExtSet (E1, x, E2)) = SOME (ExtSet (E1, x, E2))
| Deannotate _ = NONE
```

5.4.3 Parallel Evaluation

Recall that to specify that a set is to be iterated over in parallel, it is only necessary to use `<<-` instead of `<-` in a set comprehension, rewriting a CPL query $\{e \mid ..., \backslash x$ `<-` $e', ...\}$ to $\{e \mid ..., \backslash x$ `<<-` $e', ...\}$. The corresponding rule in our SYN-representation of query is a function that rewrites ExtSet to ||ExtSet and |ExtSet to |||ExtSet. Here is the straightforward ML codes implementing this rule in our system.

```
fun Parallel (|ExtSet (E1, x, E2)) =
    if WorthParallelize E1
    then SOME (|||ExtSet (E1, x, E2))
    else NONE
| Parallel (ExtSet (E1, x, E2)) =
    if WorthParallelize E1
    then SOME (||ExtSet (E1, x, E2))
    else NONE
| Parallel _ = NONE
```

Note that it is only worth while to parallelize a loop if its body contains some accesses to external data sources. Moreover, these data sources must have the ability to support at least two connections. Thus the definition of *WorthParallelize* below.

```
val WorthParallelize = SyntaxTools.Thereis (
    fn (Read(S, _, _)) ⇒ Server.GetLevel S > 1
    |_ ⇒ false)
```

5.5 Rules for Operator Migration

In this subsection, we consider the implementation of rules for migrating operations to external data servers. We use relational database servers for illustration. These rules are generally more complex because we have to deal with the nuances of external servers.

5.5.1 Projection Migration

This is the idea of rewriting a query such as $\{(\#name : n, \#age : a) \mid (\#name : \backslash n, \#age : \backslash a, ...)$ `<- process "select * from EMP" using CompanyServer` to an improved query `process "select name, age from EMP" using CompanyServer`. The projection on name and age of employees is now performed at source by the relational database server. This is an improvement for two reasons. Firstly, the relational database server can performed relational projection much more efficiently than a general engine like Kleisli. Secondly, there is now less data to transfer from the database server to our system.

A more general form of this rule is to rewrite an expression of the form $\mathsf{ExtSet}(N, x, \mathsf{Read}(S, P, M))$ to an expression $\mathsf{ExtSet}(N, x, \mathsf{Read}(S, P, M'))$, under the following circumstances. First, S must be a SQL server. Second, all occurrences of x inside the loop body N must be a projection. Third, M' is obtained from M by projecting the columns used in N. This general rule is implemented in our optimizer using the following short piece of ML codes:

```
fun PushProjection (ExtSet (N, x, Read (S, P, M))) =
    if Annotations.IsSQL S
    then case FullyProjected x N
        (* All uses of x in N are projections. Push all of them. *)
        of SOME Projs ⇒ SOME (
            ExtSet (N, x, Read (S, P,
                Unary (COString.Mk o (SQL.PushProj Projs) o COString.Km, M))))
        (* Some uses of x in N are not projections. Must not do any pushing. *)
        | NONE ⇒ NONE
    else NONE
| PushProjection _ = NONE
```

where *FullyProjected x N* tests whether all uses of x in N are projections. It returns these projections in a list *Projs*. It is implemented as follows:

```
fun FullyProjected x N =
    let (* No. of times x appears in N. *)
        val Count = ref 0
        (* The projections in N involving x. *)
        val Projs = ref []
        (* Rule to find out the projections on x in N. *)
        fun FindProjs (Variable y) = (if x = y then inc Count else (); NONE)
        | FindProjs (Proj (L, Variable y)) = (if x = y then Projs := L :: (!Projs) else (); NONE)
        | FindProjs _ = NONE
    in (* Apply FindProjs context-sensitively to N using BottomUpOnce. *)
        SimpleOptimizer.ContextSensitive FindProjs SimpleOpyimizer.BottomUpOnce N;
        if length (!Projs) = !Count
        then (* x is used only in projection. *)
            SOME (!Projs)
        else (* x is used at least once as a whole object. *)
            NONE
    end
```

There are a couple things worth noting in the implementation above. First, note the interesting use of a context-sensitive "rule" *FindProjs* to check whether x is used only in projections in the loop body N. Without this "trick," this migration rule would require a rather messy implementation. Second, note that while M is required to evaluate at runtime to an SQL expression, it can be uninstantiated at the time the rule above is applied. We have delayed the transformation of M into M' to runtime by using the Unary constructor of the SYN-representation. Should M eventually become instantiated during query optimization, the constant-folding rule presented earlier will kick in to transform it to M' during Phase 9 of the optimizer.

5.5.2 Selection Migration

This is the rewriting of a query such as $\{a \mid (\#name : \backslash n, \#age : \backslash a, ...)$ `<- process "select * from EMP" using CompanyServer`, $n = "John"\}$ to an improved query $\{a \mid (\#name : \backslash n, \#age : \backslash a, ...)$ `<- process "select * from EMP where name = 'John'" using CompanyServer`. The restriction on the name of employee to John is now performed at source by the relational database server. This is an improvement for two reasons. Firstly, the relational server can perform the selection faster, especially if it has an index on the names of employees. Secondly, there is now less data to transfer from the database server to our system.

A more general form of this rule is to rewrite an expression of the form $\mathsf{ExtSet}(N, x, \mathsf{Read}(S, P, M))$ to an expression $\mathsf{ExtSet}(N', x, \mathsf{Read}(S, P, M'))$, under the following circumstances. First, S must be a SQL server. Second, N' is obtained from N by extracting from N two lists of predicates Pve and Nve such that $N = N'$ if all predicates in Pve hold and all predicates in Nve fail. Naturally, the free variables appearing in Pve and Nve have to be amongst the free variables of N and all operations in Pve and Nve should be available in SQL. Also, each predicate in Pve and Nve must involve the loop variable x. Third, M' is obtained from M by incorporating the predicates Pve and Nve in the obvious way.

```
fun PushSelection (ExtSet (N, x, Read (S, P, M))) =
    if Annotations.IsSQL S
    then let
            (* Predicates involving these variables can be pushed. *)
            val Vars = SyntaxTools.FV N
            (* These variables must be instantiated before pushing. *)
            val Vars' = IntSet.difference (Vars, IntSet.singleton x)
            (* Extracts those predicates that can be pushed. *)
            val (Pve, Nve, N') = SyntaxTools.FlattenTests (fn E ⇒
                    (CanPushTest Vars E) andalso (IntSet.member (SyntaxTools.FV E, x))) N
        in if null Pve andalso null Nve
            then (* No predicate can be pushed. *)
                NONE
            else (* Some predicates can be pushed. *)
                SOME (ExtSet (N', x, Read (S, P, Binary (
                    fn (X, Y) ⇒ COString.Mk (SQL.PushTest Pve Nve Vars' x (COString.Km X) Y),
                    M,
                    Record (Record.MkTuple (map Variable (IntSet.listItems Vars')))))))
        end
    else NONE
  | PushSelection _ = NONE
```

The extraction of Pve, Nve, and N' from the loop body N is performed by the function *FlattenTests*. The conditions on the predicates in Pve and Nve are passed to *FlattenTests* as a check function *CanPushTests Vars*. Both are simple to implement. We give the implementation of *FlattenTests* below.

```
fun FlattenTests Check (IfThenElse (C, N, EmptySet)) =
```

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

It is worth noting that our implementation above is much more general than the example given as illustration of the rule. First, the predicates being migrated can involve variables other than the loop variable x. This allows us to migrate a predicate that expresses a join across two servers. For this reason, the rule above is actually used in Phase 12 of our optimizer, while a much simpler selection migration rule is used in Phase 6.

Second, we do not need M or any variables in Pve and Nve to be instantiated when the rule is applied. The actual transformation of M into M' is performed at runtime. We have delayed the transformation of M into M' by using the Binary constructor of the SYN-representation. Should M eventually become instantiated during query optimization, provided Pve and Nve have no free variables other than x, the constant-folding rule presented earlier will kick in to transform it to M' during Phase 13 of the optimizer. Note that if the SYN-representation does not have a facility such as Binary, we would not be able to allow Pve and Nve to contain variables other than x, severely reducing the power of this selection migration rule.

5.5.3 Join Migration

This is the rewriting of a query such as $\{(\#name : n, \#boss : m) \mid (\#name : \backslash n, \#dept : \backslash d, ...)$ `<- process "select * from EMP" using CompanyServer`, $(\#name : \backslash m, ...)$ `<- process "select * from EMP" using CompanyServer`, $(\#name : d, \#manager : m, ...)$ `<- process "select * from DEPT" using CompanyServer` $\}$ to `process "select name = e.name, boss = b.name from EMP e, EMP b, DEPT d where e.dept = d.name, b.name = d.manager" using CompanyServer`. The original query requires the employee and department tables to be sent over in their entirety many times so that the join can be performed by Kleisli. In the transformed query, the join is performed at source by the company database server, which can exploit possibly available indices on the two tables involved.

A more general form of this rule is to rewrite an expression of the form $\mathsf{ExtSet}(N, x, \mathsf{Read}(S, P, M))$ into an expression $\mathsf{ExtSet}(N', x, \mathsf{Read}(S, P, M'))$, under the following circumstances. First, S must be a

SQL server. Second, we extract from N an expression $\mathsf{ExtSet}(U, v, \mathsf{Read}(S, Q, W))$, a list of projections $ProjO$ on the variable x, two lists of predicates PO and NO that depend only on x, a list of projections $ProjI$ on the variable v, two lists of prdicates PI and NI that depend on both x and v. Essentially, $\mathsf{Read}(S, P, M)$ is the outer relation of the join and $\mathsf{Read}(S, Q, W)$ is the inner relation of the join. Every x from the outer relation that qualifies has to satisfy all of PO and fail all of NO. Every x from the outer relation and v from the inner relation has to satisfy all of PI and fail all of NI. Hence N is equivalent to `if` $\bigwedge PO \wedge \neg \bigvee NO$ `then` ExtSet(`if` $\bigwedge PI \wedge \neg \bigvee NI$ `then` U `else` $\{\}$, v, W) `else` $\{\}$. Third, N' is obtained from U by replacing all occurrences of v by x. Fourth, M' is obtained from M and W by joining them using the predicates from PO, NO, PI and NI, as well as by doing the projections in $ProjO$ and $ProjI$.

```
fun PushJoin (ExtSet (N, x, Read (S, P, M)))) =
    if Annotations.IsSQL S
    then case FullyProjected x N
        of SOME ProjO ⇒
            let val (PO, NO, N') = SyntaxTools.FlattenTests (CanPushTest (IntSet.singleton x)) N
            in case FindJoin (IntSet.singleton x) x S N'
                of SOME (PI, NI, ProjI, ExtSet (U, v, Read (_, _, W))) ⇒ SOME (
                    ExtSet (Rename x v U, x, Read (S, p, Binary (
                        fn (X, Y) ⇒ COString.Mk (SQL.PushJoin x v ProjO ProjI
                            (PO @ PI) (NO @ NI) (COString.Km X) (COString.Km Y)),
                        M,
                        W))))
                | _ ⇒ NONE
            end
        | NONE ⇒ NONE
    else NONE
  | PushJoin _ = NONE
```

The extraction of $\mathsf{ExtSet}(U, v, W)$, $ProjI$, PI, and NI from N is expressed by the function *FindJoin*, implemented below. The renaming required to obtain N' from U by replacing v by x is performed by the function *Rename*; the extremely short implementation is omitted.

```
fun FindJoin F V S (IfThenElse (M, N, EmptySet)) = (
    case FindJoin F V S N
    of SOME (Pve, Nve, Projs, ExtSet (U, v, W)) ⇒
        SOME (Pve, Nve, Projs, ExtSet ( IfThenElse (M, U, EmptySet), v, W))
    | _ ⇒ NONE
  | FindJoin F V S (IfThenElse (M, EmptySet, N)) = (
    case FindJoin F V S N
    of SOME (Pve, Nve, Projs, ExtSet (U, v, W)) ⇒
        SOME (Pve, Nve, Projs, ExtSet ( IfThenElse (M, EmptySet, U), v, W))
    | _ ⇒ NONE
  | FindJoin F V S (ExtSet (M, x, N)) =
    let val Join = (case N
```

```
        (* Inner relation N is from external DB. May be can push. *)
        of Read (S', p, W) ⇒
            if S = S' andalso IntSet.isEmpty ( IntSet.intersection (F, SyntaxTools.FV N))
            then (* N is from same DB as outer relation and does not depend on outer loop variable. *)
                case FullyProjected x M
                of SOME Projs ⇒ (
                    case SyntaxTools.FlattenTests (CanPushTest (IntSet.addList (IntSet.empty, [x, V]))) M
                    of ([], [], _) ⇒ NONE
                    | (Pve, Nve, U) ⇒ SOME (Pve, Nve, Projs, ExtSet (U, x, N)))
                | NONE ⇒ NONE
            else (* N not from same DB as outer relation or depends on outer loop variable. *)
                NONE
        (* Inner relation N is not external DB. No need to push. *)
        | _ ⇒ NONE)
    in case Join
        of NONE ⇒ (case FindJoin (IntSet.add (F, x)) V S M
            of SOME (Pve, Nve, Projs, ExtSet (U, v, W)) ⇒
                    SOME (Pve, Nve, Projs, ExtSet (ExtSet (U, x, N), v, W))
            | _ ⇒ NONE
        | _ ⇒ Join
    end
  | FindJoin _ _ _ _ = NONE
```

5.6 Rules Using Meta Information

In this subsection, we describe a rule that uses meta information to reorder joins involving two external relational databases. Note that we do not need to reorder joins involving the same database, since such joins would be migrated directly to that database.

We first give an example to motivate the need for meta information. Suppose we want to find the salary of the manager of the Sales Department, if his age is 25. For the sake of illustration, suppose employee information is kept on the database in ServerA but department information is kept on the database in ServerB. One possible query is: $\{s \mid (\#name : \backslash x, \#salary : \backslash s, ...)$ `<- process "select * from Emp a where a.age = 25" using ServerA`, $(\#manager : x, ...)$ `<- process "select * from Dept b where b.name = 'Sales'" using ServerB` $\}$. A second possibility is $\{s \mid (\#manager : \backslash x, ...)$ `<- process "select * from Dept b where b.name = 'Sales'" using ServerB`, $(\#name : x, \#salary : \backslash s, ...)$ `<- process "select * from Emp a where a.age = 25" using ServerA` $\}$. The first query obtains from ServerA all those employees who are 25 years old, then tests them one-by-one on ServerB to see who runs the Sales Department. The second query obtains the Sales manager from ServerB first, then checks his age on ServerA.

Suppose there are m employees aged 25 and n departments named Sales. The first alternative means ServerB will be accessed m times while ServerA wll be accessed once. The second alternative means ServerA will be accessed n times while ServerB will be accessed once. In real life, m is usually a big number while n is usually equal to 1. So the second alternative is much better than the first.

Unfortunately, there is no way to decide on the basis of the syntax of these two queries whether $m > n$ is true. Some meta information must be made available in order for the optimizer to make an intelligent decision in the above situation.

The above kind of optimization corresponds to rewriting the first alternative to the second alternative. A more general form is to rewrite an expression of the form $\mathsf{ExtSet}(N, x, \mathsf{Read}(S, P, M))$ to an expression $\mathsf{ExtSet}(\mathsf{ExtSet}(U, x, \mathsf{Read}(S, P, M)), y, V)$, under the following circumstances. First, S must be a SQL server. Second, M must already be instantiated to an SQL command. Third, the "selectivity" of M can somehow be determined using some meta information to be m. Fourth, N can be shown to be equivalent to $\mathsf{ExtSet}(U, y, V)$ such that V does not depend on x and has selectivity less than m. This general rule is implemented in our optimizer as follow:

```
fun Reorder (ExtSet(N, x, Read(S, P, String M))) =
  if Annotations.IsSQL S
  then let
      (* Extract user and name of database *)
      val [DB, Usr] = Annotations.GetSQLDB Sv
      (* Estimate selectivity using meta info *)
      val m = SQLDict.ExtractSelectivity DB Usr M
    in case MoveItOut m N
      (* Found inner relation with smaller selectivity *)
      of SOME (ExtSet(U, y, V)) ⇒
        SOME (ExtSet(ExtSet(U, x, Read(S, P, String M)), y, V))
      | _ ⇒ NONE
    end
  else NONE
| Reorder _ = NONE
```

The transformation of N to $\mathsf{ExtSet}(U, y, V)$ is expressed by the function *MoveItOut*, implemented below. It simply looks for the first inner relation having selectivity smaller than m.

```
fun MoveItOut m (IfThenElse(E1, E2, EmptySet)) = (
  case MoveItOut m E2
  of SOME (ExtSet(E, x, R)) ⇒ SOME (ExtSet(IfThenElse(E1, E, EmptySet), x, R))
  | _ ⇒ NONE)
| MoveItOut m (IfThenElse(E1, EmptySet, E2)) = (
  case MoveItOut m E2
  of SOME (ExtSet(E, x, R)) ⇒ SOME (ExtSet(IfThenElse(E1, EmptySet, E), x, R))
  | _ ⇒ NONE)
| MoveItOut m (ExtSet(E, x, Read(S, P, String M))) =
  if Annotations.IsSQL S
  then let val [DB, Usr] = Annotations.GetSQLDB S
    val n = SQLDict.ExtractSelectivity DB Usr M
    in if n < m
      then SOME (ExtSet(E, x, Read  (S, P, String M)))
```

```
        else NONE
      end
    else NONE
| MoveItOut _ _ = NONE
```

6 Conclusion

This report describes the extensible optimizer of the Kleisli/CPL system. It has multiple phases. Each phase has a rule base and a rule application strategy. A simple method for generating rule application strategies is given and is illustrated with many example strategies actually used by our system. A wide variety of example rules actually used in our system are given to illustrate the simple way in which we can implement optimization rules. The rules include vertical loop fusion, filter promotion, constant folding, invariant code motion, evaluation mode assigment, access priority assignment, join migration, and so on.

We originally planned to build an "optimizer generator," in the same sense as a "compiler generator" like Yacc, for our system. However, it is evident from the examples in this report an optimizer generator like [8] or [12] would probably not simplify thing very much.

The idea of rule-based optimization and extensible optimizer is not new [8, 12, 3, 10, 19, etc.] However, it is worth pointing out that this report stands out in several ways. First, it presents a simple technique for generating rule application strategies. Even though rule application strategies constitute an important part of any rule-based optimizer, techniques for generating them were seldom discussed in these earlier papers.

Second, it presents a simple representation for a useful class of query languages and this representation enables us to implementation rules in a very simple way. In fact, sufficiently simple that we can write down the actual codes of these rules in a few lines. Even though query representation and rules are a fundamental part of any rule-based query optimizer, their implementation details were seldom discussed in these papers.

Finally, it deals with queries involving nested collection types, multiple remote heterogeneous data servers, imported operators, and so on. More interestingly, despite the fact that these queries are more complex than those a relational database optimizer has to deal with, the optimization rules are relatively simple.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, Chichester, England, 1987.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[3] L. Becker and R. H. Gueting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17:247–303, 1992.

[4] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transaction on Programming Languages and Systems*, 6(4):487–504, October 1984.

[5] Peter Buneman, Susan Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A data transformation system for biological data sources. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, August 1995.

[6] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[7] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[8] Leonidas Fegaras, David Maier, and Tim Sheard. Specifying rule-based query optimizers in a reflective framework. In *Proceedings of 3rd International Conference on Deductive and Object Oriented Databases*, Scottsdale, Arizona, December 1993.

[9] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, England, 1988.

[10] Johann Christoph Freytag. A rule based view of query optimization. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 173–180, San Francisco, May 1987.

[11] A. Goldberg and R. Paige. Stream processing. In *Proceedings of ACM Symposium on LISP and Functional Programming*, pages 53–62, Austin, Texas, August 1984.

[12] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, November 1987. Available as University of Wisconsin Computer Sciences Technical Report 724.

[13] Kyle Hart, Limsoon Wong, Chris Overton, and Peter Buneman. Using a query language to integrate biological data. In *Abstracts of 1st Meeting on the Interconnection of Molecular Biology Databases*, Stanford, August 1994. Available at `http://www.cis.upenn.edu/~cbil/mimbd94/mimbd94CPL.html`.

[14] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 155–166, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-93-95.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[16] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

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

[18] Anne Ngu, Lingling Yan, and Limsoon Wong. Heterogeneous query optimisation using maximal subqueries. In Songchun Moon and Hideto Ikeda, editors, *Proceedings of 3rd International Symposium on Database Systems for Advanced Applications, Taejon, Korea, April, 1993*, pages 413–420, April 1993.

[19] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible rule-based query rewrite optimization in Starburst. *SIGMOD Record*, 21(2):39–48, June 1992.

[20] Didier Remy. Efficient representation of extensible records. In P. Lee, editor, *Proceedings of ACM SIGPLAN Workshop on ML and its Applications*, pages 12–16, 1992.

[21] John Reppy. Concurrent programming with events: The Concurrent ML manual. Technical report, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, February 1993.

[22] Dan Suciu and Jan Paredaens. Any algorithm in the complex object algebra needs exponential space to compute transitive closure. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 201–209, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-94-04.

[23] P. W. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, pages 49–62. Morgan Kaufmann, August 1991.

[24] P. W. Trinder and P. L. Wadler. Improving list comprehension database queries. In *Proceedings of TENCON'89, Bombay, India*, pages 186–192, November 1989.

[25] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies*. Computer Science Press, Rockvill, MD 20850, 1989.

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

[27] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. *Journal of Computer and System Sciences*, ???(???):???–???, ??? To appear.

[28] Limsoon Wong. *The Collection Programming Language Reference Manual*. Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, October 1995. Available at `http://sdmc.iss.nus.sg/kleisli/psZ/cpl-defn.ps`.

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

[30] Limsoon Wong. *The Kleisli Query System Reference Manual*. Institute of Systems Science, Heng Mui Keng Terrace, Singapore 0511, October 1995. Available at `http://sdmc.iss.nus.sg/kleisli/psZ/kleisli-manual.ps`.