The Functional Guts of the Kleisli System

Limsoon Wong

Kent Ridge Digital Labs
Singapore

Wong, JFP 10:19–56, 2000.

1 Introduction

A slide show from Kris Informatics Inc.

2 Quick Tour of Kleisli

2.1 Complex Object Types

- Data model of complex objects:

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

- Eg., the feature table of GenPept report has type:

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

- Here is feature table from GenPept report 131470:

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

- Why such a data model/type system? Because:
  - Relational/deductive databases imposes 1-NF. Efficient to implement, but...

    ⇒ difficult to map external data sources into 1-NF, and tends to split objects into hundreds of pieces.

  - Many programming languages such as C, SML, etc. insist on creating new types and do not allow “free-forming” type constructors. Rich, but...

    ⇒ not suitable for database queries because
    * almost every query results in a “new” type
    * some input/output schemas take pages to write down

2.2 Collection Programming Language

- Extract the titles and features of those elements of `DB` whose titles contain `tyrosine` as a substring.

```
{ (#title: x.#title, #feature: x.#feature)
| \x <- DB, x.#title string-islike "%tyrosine%" };
```

- Flatten `DB` completely.

```
{(#title:x.#title, #feature:f.#name,
  #start:f.#start, #end:f.#end,
  #anno-name:a.#anno_name, #anno-descr:a.#descr)
| \x <- DB, \f <- x.#feature, \a <--- f.#anno};
```

- A "group-by" query.

```
let \DB' == {(#entry:x, #organism:a.#descr)
   | \x <- DB, \f <- x.#feature, \a <--- f.#anno,
     a.#anno_name = "organism"} in
let \ORG == {y.#organism | \y <- DB'}
in [(#organism:z, #entries: {v.#entry
       | \v <- DB', v.#organism = z})
   | \z <--- [u | \u <- ORG]];
```

2.3 Expressive Power: Is More Better?

- Programming languages want Turing completeness.

- Query languages want to make sure that everything expressible is also practical.
  - Turing completeness is harmful
  - Queries should be in a low complexity class
  - Preferrably nothing worse than $O(n \cdot \log n)$
  - Queries should be such that the optimizer has a good chance to optimize them to no more than $O(n \cdot \log n)$

- So query languages are restricted programming languages

- The core of CPL has its roots in $\mathcal{NRC}$, a nested relational calculus obtained by a restriction on structural recursion.

2.4 Theoretical Roots

- Syntax of $\mathcal{NRC}$

$$\frac{}{x^t : t} \qquad \frac{e_1 : t \quad e_2 : t}{e_1 = e_2 : \mathcal{B}}$$

$$\frac{}{\mathit{true} : \mathcal{B}} \qquad \frac{}{\mathit{false} : \mathcal{B}} \qquad \frac{e : \mathcal{B} \;\; e_1 : t \;\; e_2 : t}{\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : t}$$

$$\frac{e_1 : t_1 \quad e_2 : t_2}{(e_1, e_2) : t_1 \times t_2} \qquad \frac{e : t_1 \times t_2}{\pi_1\ e : t_1} \qquad \frac{e : t_1 \times t_2}{\pi_2\ e : t_2}$$

$$\frac{e : t}{\{e\} : \{t\}} \qquad \frac{e_1 : \{t\} \;\; e_2 : \{t\}}{e_1 \cup\ e_2 : \{t\}} \qquad \frac{}{\{\}^t : \{t\}}$$

$$\frac{e_1 : \{t_1\} \;\; e_2 : \{t_2\}}{\bigcup\{e_1 |\ x^{t_2} \in e_2\} : \{t_1\}}$$

  where $\bigcup\{e_1 |\ x^{t_2} \in e_2\} = f(o_1) \cup \cdots \cup f(o_n)$, if $f(x) = e_1$ and $e_2 = \{o_1, \ldots, o_n\}$.

- **Theorem 1** $\mathcal{NRC} = \mathit{Thomas\&Fischer} = \mathit{Schek\&Scholl} = \mathit{Colby}$

Buneman&Naqvi&Tannen&Wong, TCS 149:3–48, 1995.

- A rewrite system.
  - $\pi_i(e_1, e_2) \leadsto e_i$
  - *if true then* e_2 *else* $e_3 \leadsto e_2$
  - *if false then* e_2 *else* $e_3 \leadsto e_3$
  - $\pi_i($*if* e *then* e_2 *else* $e_3) \leadsto$ *if* e *then* $\pi_i\, e_2$ *else* $\pi_i\, e_3$
  - $\bigcup\{e \mid x \in e_1 \cup e_2\} \leadsto \bigcup\{e \mid x \in e_1\} \cup \bigcup\{e \mid x \in e_2\}$
  - $\bigcup\{e \mid x \in \{\}\} \leadsto \{\}$
  - $\bigcup\{e \mid x \in \{e'\}\} \leadsto e[e'/x]$
  - $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \leadsto \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$
  - $\bigcup\{e \mid x \in$ *if* e_1 *then* e_2 *else* $e_3\} \leadsto$ *if* e_1 *then* $\bigcup\{e \mid x \in e_2\}$ *else* $\bigcup\{e \mid x \in e_3\}$

- **Proposition 2** *The rewrite system above is sound and strongly normalizing.*

- **Theorem 3** *Let* $e : s$ *be an expression of* $\mathcal{NRC}$ *in normal form. Then* $ht(e) \leq \max(\{ht(s)\} \cup \{ht(t) \mid s$ *is the type of a free variable in* $e\})$. *Thus,* $\mathcal{NRC}$ *has the conservative extension property.*

- **Corollary 4** $\mathcal{NRC} = FO(=) =$ *RelationalAlgebra over flat relations.*

Wong, JCSS 52:495–505, 1996.

- $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$, an extension to capture SQL.

Add the following constructs to $\mathcal{NRC}$:

$$\frac{e_1 : \mathcal{Q} \quad e_2 : \mathcal{Q}}{e_1 + e_2 : \mathcal{Q}} \qquad \frac{e_1 : \mathcal{Q} \quad e_2 : \mathcal{Q}}{e_1 \cdot e_2 : \mathcal{Q}} \qquad \frac{e_1 : \mathcal{Q} \quad e_2 : \mathcal{Q}}{e_1 \div e_2 : \mathcal{Q}}$$

$$\frac{e_1 : \mathcal{Q} \quad e_2 : \mathcal{Q}}{e_1 - e_2 : \mathcal{Q}} \qquad \frac{e_1 : \mathcal{Q} \quad e_2 : \{s\}}{\Sigma\{\!|e_1 \mid x^s \in e_2|\!\} : \mathcal{Q}} \qquad \frac{e_1 : \mathcal{Q} \quad e_2 : \mathcal{Q}}{e_1 \leq e_2 : \mathcal{B}}$$

where $\Sigma\{\!|e_1 \mid x \in e_2|\!\} = f(o_1) + \ldots + f(o_n)$, if $f(x) = e_1$ and $e_2 = \{o_1, \ldots, o_n\}$.

- **Theorem 5** $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ *has the conservative extension property.*

- **Claim 6** $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ *over flat relations is "entry-level" SQL.*

Libkin&Wong, JCSS, 55:241–272, 1997.

- **Theorem 7** *Let* $P : \mathcal{Q} \to \mathcal{B}$ *be a predicate definable in* $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$. *Then either* P *holds for finitely many natural numbers or* P *fails for finitely many natural numbers. Hence a predicate on natural number is definable in* $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ *iff it is finite-cofinite.*

Proof. $P : \mathcal{Q} \to \mathcal{B}$ has height 0. By the conservative extension property on $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$, any implementation of it in $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ is equivalent to one that does not use sets. It is easy to see that such an implementation must be equivalent to something like:

$$\lambda n.(\bigvee \bigwedge p_i(n) \leq 0)$$

where $p_i(n)$ are polynomials in n. The result now follows by the fact that polynomials have finite number of roots.

Libkin&Wong, JCSS 55:241–272, 1997.

- **Definition 8** *A language has the bounded degree property if, for any graph query Q in it, there is a function $f_Q : \mathcal{N} \to \mathcal{N}$ such that, for any graph G of degree k, the number of distinct degrees in $Q(G)$ is at most $f_Q(k)$.*

- **Proposition 9** *$\mathcal{NRC}$ has the bounded degree property.*

- **Theorem 10** *$\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ has the bounded degree property.*

  Proof. Let $Q(x)$ be a graph query in normal form in $\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ and k be a natural number. Then there are boolean conditions $p_1(x)$, ..., $p_d(x)$ and $e_1(x)$, ..., $e_{d+1}(x)$ are $\mathcal{NRC}$ expressions, such that for any graph x of degree at most k, it is the case that

  $$Q(x) = \textit{if } p_1(x) \textit{ then } e_1(x) \textit{ else } \ldots \textit{ else } e_{d+1}(x)$$

  The theorem now follows from the bounded degree property of $\mathcal{NRC}$.

- **Corollary 11** *$\mathcal{NRC}(\mathcal{Q}, +, \cdot, -, \div, \Sigma, \leq^{\mathcal{Q}}, =)$ cannot express transitive closure, balanced binary tree test, etc.*

Dong&Libkin&Wong, TCS 239:277–308, 2000.

3 Influence of Functional Programming

3.1 Type Inference and Self-Describing Exchange Format

3.1.1 Impact of Dynamic Heterogeneous Environment

- Dynamic heterogeneous environment means
  - Many databases, mand softwares
  - No explicit schemas, extremely large schemas
  - Adhoc combination, constant evolution

- Any good integration system for such an environment must
  - compile queries based on their structure
  - have data model that external sources map easily to
  - shield queries from evolution of sources
  - have exchange format that is easy to use

3.1.2 Flexible Type Inference

- Kleisli has ML-style polymorphic type system.
  ⇒ compiles queries based on their structure.

- Sets, bags, lists, records, and variants can be freely combined.
  ⇒ has data model that external sources map easily to.

- Kleisli has Machiavelli-style polymorphic record types. Eg.,

  ```
  \R => {x.#name | \x <- R, x.#salary > 1000}
  ```

  ⇒ allows data sources to evolve by adding fields to records.

- Kleisli has full-fledged variant types. Eg.,

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

  ⇒ allows data sources to evolve by adding variants.

- *A question to the audience: Why do FP languages such as ML and Haskell lack first-class variant types? Such types are crucial for dealing with standards such as ASN.1.*

- Kleisli does take some pragmatic risks on type safety. Eg.,

```
primitive S1 == \x1 => scop (x1);
primitive S2 == \x2 => scop (<#sidinfo:x2>);
primitive S3 == \x3 => scop (<#numsid:x3>);
```

The type produced for `S1` is

```
<#sidinfo:string, #sidallinfo:unit,  ...,
 #numsid:string, #pdbsid:string> -> {''3}
```

The type produced for `S2` is

```
string -> {(
 #sid:string, #pdb:string, #num:string, ...,
 #type: (#cl:string, #cf:string, ...),
 #desc: (#cl:string, #cf:string, ...))}
```

The type produced for `S3` is

```
string -> {string}
```

Kleisli uses meta information and context information to improve type safety whenever it can. If not possible, it falls back on runtime checking.

3.1.3 Self-Describing Data Exchange Format

- For external systems to exchange data with Kleisli
- Each data type has unambiguous encoding
- Each data item can be parsed without reference to schema
- No more than a few characters look-ahead are needed
- Direct correspondence to complex object data model
- Here is the salient parts of the exchange format

Data Type	Data Layout	Remarks
Records	(#l_1: O_1, $\vdots$, #l_n: O_n)	Record is put inside round brackets. The label-value triplets enumerate the fields of the record
Variants	<#l: O>	Variant is put inside angle brackets
Sets	{ O_1, $\vdots$, O_n}	Set is put inside curly brackets

- There is also a compressed form which average about 70% more efficient

Wong, BIBE 2000, to appear.

3.2 Kleisli Triples and Abstract Syntax

- The internal representation of monads by Kleisli is

$$\begin{aligned} ext(f)\{\} &= \{\} \\ ext(f)\{o\} &= f(o) \\ ext(f)(A\ U\ B) &= ext(f)(A)\ U\ ext(f)(B) \end{aligned}$$

  $ext(f)(R)$ where $f(x) = e$ is `ext{ e | \x <- R}` in CPL.

- Kleisli does not use comprehension-style represention of monads, except as surface syntax.

- Surface syntax: the easier to read the better. Eg.,

```
{ (x, y) | \x <- R, \y <- S }
```

vs

```
ext{ ext{ {(x, y)} | \y <- S} | \x <- R}
```

- So comprehension syntax is good for surface syntax. But consider the vertical loop fusion optimization:

```
      { e | G1, ..., Gn,
            \x <- { e' | H1, ..., Hm},
           J1, ..., Jk}

  ==> { e[e'/x] | G1, ..., Gn,
                  H1, ..., Hm,
                  Ji[e'/x], ..., Jk[e'/x] }
```

The rule is easy to grasp. But to properly implement the informal ..., you will need a cumbersome traversal routine.

- Abstract syntax: the easier to analyse the better. The Kleisli representation wins hands down. Eg., the vertical loop fusion becomes:

```
      ext{ e1 | \x <- ext { e2 | \y <- e3}}
  ==> ext{ ext { e1 | \x <- e2} | \y <- e3}
```

  The troublesome ... no longer appears. Such a rule can be coded up straightforwardly in almost any implementation language.

- Let's see the detail in Kleisli.

```
(*
 * Vertfusion : SYN -> SYN option
 *)

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

3.3 Higher-Order Functions and Optimization

3.3.1 Overview

- Demands of the Kleisli optimizer:
  - It has a number of phases.
  - Each phase has a rule base and a rule-application strategy.
  - It supports a large number of strategies.

- Using higher-order functions, these rule-application strategies can be decomposed into
  - "Traversal" component common to all strategies
  - "Control" component specialized to individual strategy

- So higher-order functions can generate the strategies extremely simply, resulting in a small optimizer core

3.3.2 Example Phases and Strategies

Phase	Optimization	Strategy
1	De-annotation, Strip lazy and parallel tags	BottomUpOnce
2	Monad optimization. Eg. pipelining	BottomMany
3	Join migration	BottomUpOnce
4	Selection migration	TopDownOnce
5	Projection migration	TopDownOnce
6	“Dynamic” selection migration	TopDownOnce
7	Monad optimization again	BottomUpMany
8	Specialize polymorphic ops to monomorphic ones	BottomUpOnce
9	Replace Remy’s polymorphic projection over homogeneous collection by a memoised version to amortize offset computations	EntireOnce
10	Re-annotation. Insert lazy and parallel tags	BottomUpOnce

3.3.3 Common "Traversal" Component

- "Traversal" component is a higher-order function:

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

- Given a rewrite rule `r` and a query `Q`, it nonrecursively rewrites immediate substrees in `Q` using `r`.

```
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)
| ...
```

3.3.4 Simple "Control" Components

- Assume a rule-application strategy `S` is a function of type

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

- Assume

  ```
  val Pick: RULEDB -> SYN -> SYN option
  ```

  is a function that takes a rule base `R` and a query `Q` and returns `NONE` if no rule in `R` is applicable, and `SOME Q'` if some rule in `R` can be applied to rewrite `Q` to `Q'`.

- “Control” component of MaxOnce strategy.

  Applies rules to maximal subtrees; applies each rule at most once along each path from the root to the leaves.

```
fun MaxOnce RDB Qry =
  case Pick RDB Qry
  of SOME ImprovedQry => ImprovedQry
  | NONE => Decompose (MaxOnce RDB) Qry
```

- “Control” component of BottomUpOnce strategy.

  Applies rules in a leaves-to-root pass; rewrites each node at most once.

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

3.3.5 More Complex "Control"

- How do you implement a rule such as

```
    if E1 then ...  E1  ... else E3
==> if E1 then ... true ... else E3
```

- Such a rule really has two parts
  - Inner part just rewrites `E1` to `true`.
  - Outer part identifies the context and scope of the inner part: the `then`-branch

and the informal `...` are again in the way!

- A rule base `RDB` is represented in Kleisli as

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

  where

  - `Rules` stores the list of rules of that rule base
  - `Trace` stores a function `f` used for "tracing" the usage of rules in the rule base.
  - `DoTrace` stores a flag to indicate whether tracing is to be done

- If tracing is indicated, whenever a rule `N` in the rule base is applied successfully to transform a query `Q` to `Q'`, the trace function is invoked as `f N Q Q'` to record a trace.

- Typically this means a message like "`Q` is rewritten to `Q'` using rule `N`" is printed.

- But trace functions can be made to do much more, such as implementing the If-then-else absorption rule mentioned earlier. Here is the rule:

```
fun OuterPart (IfThenElse(Bool _,_,_)) = NONE
| OuterPart (IfThenElse(E1,E2,E3)) =
   let fun InnerPart E =
         if SyntaxTools.Equiv E1 E
         then SOME(Bool true)
         else NONE
   in (*
       * apply InnerPart using the TopDownOnce
       * strategy inside scope of then-branch
       * using the ContextSensitive rule generator.
       *)
      case ContextSensitive InnerPart TopDownOnce E2
      of SOME E2' => IfThenElse(E1,E2',E3)
      |  NONE => NONE end
| OuterPart _ = NONE
```

- It uses the `ContextSensitive` rule generator:

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

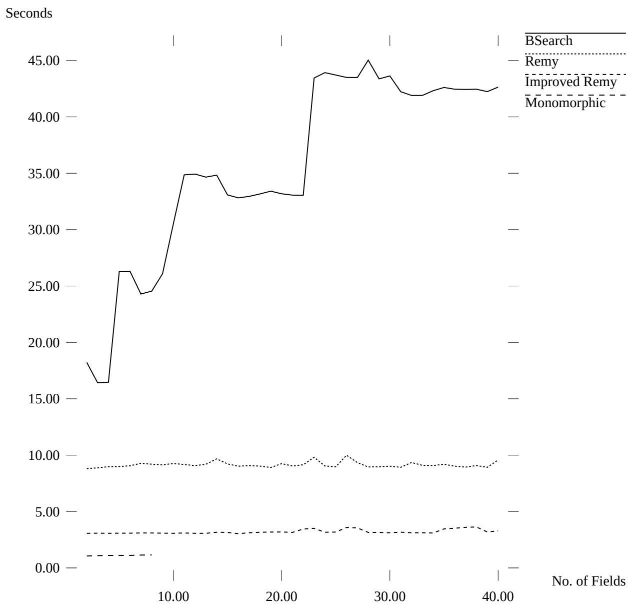
- The `ContextSensitive` rule generator is also used for migrating selections, projections, joins, etc. to external relational database systems.

3.4 Constant-Time Polymorphic Record Projection

- Kleisli uses an idea of Remy for constant-time projection
- Kleisli optimizes some Remy projections into jumps
- Effect of turning Remy projections into jumps on:

```
{(#name:n, #age:a, #sex:s) |
  (#name:\n, #age:\a, ...) <- DB1,
  (#name: n, #sex:\s, ...) <- DB2}
```

with respect to changes in the number of fields in `DB1` and `DB2`:



2nd curve: before optimization. 3rd curve: after optimization.

Wong, SIGMOD Record 24:34–39, 1995.

4 Impact on Bioinformatics

4.1 A DOE "Impossible" Query

Find for each gene located on a particular cytogenetic band of a particular human chromosome, as many of its non-human homologs as possible.

4.2 Data Sources Involved

source	type	location	remarks
GDB	Sybase	Baltimore	flat tables SQL-based joins cytogenetic bands info
Entrez	ASN.1	Bethesda	nested tables keyword and links seq and homolog info

4.3 Implementation in Kleisli/CPL

```
sybase-add (#name:"gdb", ...);
readfile locus from "locus_cyto_location" using gdb;
readfile eref from "object_genbank_eref" using gdb;

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

- On the surface, it fetched the `locus` table in its entirety once and the `eref` table in its entirety n times from GDB, as a naive evaluation would be two nested loops over these tables.

- In reality, the Kleisli optimizer was able to migrate the join and projections on these two tables into a single efficient access to GDB. It also parallelized and pipelined the access to Entrez.

Davidson&Overton&Tannen&Wong, IJDL 1:36–53, 1997.

4.4 Quotable Quotes

- D. Benton:

  Recent implementations (eg. Kleisli) have shown it possible to provide access both to relational databases and unconventional sources of biological data without building a monolithic database or writing a very application-specific code for each query.

- P.Karp:

  An example of a complex query could be answered using the Kleisli system.

- N.Murgolo:

  Kleisli was quite impressive in this operation as in about an hour it produced the desired result compared to 2 days with a similar [Competitor X] procedure.

- J.Greene:

  I have successfully ported one of my most time-consuming [Competitor Y] jobs to Kleisli and am amazed at how much faster it is. It is incredibly valuable to have improved the performance of that job as well as to have learned to do some things in Kleisli.

5 Conclusions

- Kleisli deploys functional programming technologies.
- Kleisli has an impact on bioinformatic data integration.
- Key reasons of Kleisli's success are:

FP-related	non-FP
Complex object data model	Large number of drivers
High-level query language	
Self-describing data exchange format	
Good query optimization	
Implemented on top of SML	

6 Acknowledgements

- Early collaborators:
  Peter Buneman, Leonid Libkin, Val Tannen, Dan Suciu.

- Product development:
  Jing Chen, Jiren Wang, Oliver Wu.

- Introduction to bioinformatics:
  Chris Overton, Susan Davidson.

- Education in molecular biology:
  Anthony Ting, Nam-Hai Chua, Daphna Strauss.