THE COLLECTION PROGRAMMING LANGUAGE
Reference Manual

Limsoon Wong

Kent Ridge Digital Labs
21 Heng Mui Keng Terrace, Singapore 119613
`email: limsoon@krdl.org.sg`

5 May 1999

Version D Release 5 May 1999

Contents

List of Figures

1 Introduction

1.1 Introduction

Based on some of the ideas described in [6], I have built a prototype query system called Kleisli. (See *The Kleisli Query System Reference Manual.*) The system is designed as a database engine to be connected to the host programming language ML [23] via a collection of libraries of routines. These routines are parameterized for more open and better control so that expert users do not have to resort to wily evasion of restriction in their quest for performance. (There is strong evidence [29] that experts demonstrate a canny persistence in uncovering necessary detail to satisfy their concern for performance.)

I have included an implementation of a high-level query language, for non-expert users, called CPL with the prototype. CPL stands for *Collection Programming Language.* The libraries actually contain enough tools for a competent user to quickly build his own query language or command line interpreter to use in connection with Kleisli and the host language ML. In fact, CPL is an example of how to use these tools to implement query languages for Kleisli. This report is intended as an informal but accurate description of CPL.

1.2 Organization

Section 2. A rich data model is supported in CPL. In particular, sets, lists, bags, records, and variants can be freely combined. The language itself is obtained by orthogonally combining constructs for manipulating these data types. The data types and the core fragment of CPL is described in this section. Examples are provided to illustrate CPL's modeling power.

Section 3. A comprehension syntax is used in CPL to uniformly manipulate sets, lists, and bags. CPL's comprehension notation is a generalization of the list comprehension notation of functional languages such as Miranda [39]. The comprehension syntax of CPL is presented in this section and its semantics is explained in terms of core CPL. Examples are provided to illustrate the uniform nature of list-, bag-, and set-comprehensions.

Section 4. A pattern matching mechanism is supported in CPL. In particular, convenient partial-record patterns and variable-as-constant patterns are supported. The former is also available in languages such as Machiavelli [26], but not in languages such as ML [23]. The latter feature is not available elsewhere at all. The pattern matching mechanism of CPL is presented in two stages. In the first stage, simple patterns are described. In the second stage, enhanced patterns are described. Semantics is again given in terms of core CPL. Examples are provided to illustrate the convenience of pattern matching.

Section 5. More examples are given to illustrate other features of CPL. These features include: (1) Types are automatically inferred in CPL. In particular, CPL has polymorphic record types. However, the type inference system is simpler than that of Ohori [27], Remy [30], etc. (2) External functions can be easily imported from the host system into CPL. Scanners and writers for external data can be easily added to CPL. More details can be found in *The Kleisli Query System Reference Manual.* (3) An extensible optimizer is available. The basic optimizer does loop fusion, filter promotion, and code motion. It optimizes scanning and printing of external files. It has been extended to deal with joins by picking alternative join operators and by migrating them to external servers. More details can be found in *The Kleisli Query System Reference Manual* and in [41, 4].

Section 6. CPL supports a limited form of subtyping. This subtyping is restricted to user-defined types that have no parameters. It can be used to emulate single inheritance. Examples are provided to illustrate this mechanism.

1.3 Acknowledgement

I would like to thank Peter Buneman, Leonid Libkin, Val Tannen, and Dan Suciu. They greatly influenced the design of CPL.

2 The Core of CPL

I first describe CPL's types. Then I describe the core fragment of CPL. The core fragment is based on the central idea of restricting structural recursion to homomorphisms of collection types. In fact, when restricted to sets, CPL is really a heavily sugared version of $\mathcal{NRC}(bool, =)$, the abstract nested relational calculus described in [6]. An alternative and less formal introduction to a variant of CPL can be found in [5]. Lastly, several examples are provided to illustrate CPL's modeling power.

2.1 Types

The ground complex object types, ranged over by s and t, are given by the grammar below. The l_i are labels and are required to be distinct. The b ranges over base types and parameterized user-defined types. Parameterized user-defined types are really a sort of abstract data types, and for the purpose of this report can be treated as base types.

$$s ::= b \mid \{s\} \mid \{|s|\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : s_1, \cdots, l_n : s_n\texttt{)} \mid \texttt{<}l_1 : s_1, \cdots, l_n : s_n\texttt{>}$$

The ground types, over which u and v range, are given by the grammar below.

$$u ::= b \mid \{s\} \mid \{|s|\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)} \mid \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \mid u \texttt{ -> } v$$

CPL allows $\texttt{(}u_1, \cdots, u_n\texttt{)}$ as a syntactic sugar for $\texttt{(}\#1 : u_1, \cdots, \#n : u_n\texttt{)}$. Labels in CPL always start with the #-sign.

Objects of type $\{s\}$ are finite sets whose elements are objects of type s. Objects of type $\{|s|\}$ are finite bags whose elements are objects of type s. Objects of type $\texttt{[}s\texttt{]}$ are finite lists whose elements are objects of type s. Objects of type $\texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)}$ are records having exactly fields l_1, ..., l_n and whose values at these fields are objects of types u_1, ..., u_n respectively. An object of type $\texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>}$ is called a variant object and is a pair $\texttt{<}l_i : o\texttt{>}$ such that o is an object of type u_i; that is, it is an object of type u_i tagged with the label l_i. (Variants are also called tagged-unions and co-products. See Gunter [15] or Hull and Yap [18] for more information.) Objects of type u `->` v are functions from type u to type v. Included amongst the base types are `num`, `string`, `bool`, and `unit` (which is the type having exactly the empty record `()` as its only object). It is also possible to extend CPL with new base types; see *The Kleisli Query System Reference Manual.*

Observe that function types u `->` v in CPL are higher-order because u and v can contain function types. This is different from the first-order function types $s \to t$ of $\mathcal{NRC}(bool, =)$ in [6]. Higher-order function types are allowed in CPL for two reasons. To discuss these two reasons, I need some results obtained by Suciu and myself [34] on the two forms of structural recursion *sri* and *sru*. Let $\mathcal{HNRC}(bool, =, sri)$ and $\mathcal{HNRC}(bool, =, sru)$ respectively denote the language obtained by generalizing $\mathcal{NRC}(bool, =, sri)$ and $\mathcal{NRC}(bool, =, sru)$ to higher-order function types. We showed that $\mathcal{HNRC}(bool, =, sri) = \mathcal{HNRC}(bool, =, sru) = \mathcal{NRC}(bool, =, sri) = \mathcal{NRC}(bool, =, sru)$ over the class of first-order functions. Hence every function of type $s \to t$ expressible using the higher-order languages is also expressible using the first-order languages. This result gives us the first reason for having higher-order function types in CPL— it makes many things more convenient but without making analysis of first-order expressive power of CPL more complicated. Another result in the same paper [34] is that all uniform implementations of *sri* in $\mathcal{NRC}(bool, =, sru)$ are bound to be expensive while there are efficient uniform implementations of *sri* in $\mathcal{HNRC}(bool, =, sru)$. This gives us the second reason for having higher-order function types in CPL— it allows the implementation of more efficient algorithms. See my paper with Suciu [34] for details.

CPL also has nonground types. I only intend to explain how to read a CPL type expression having nonground types below. The nonground complex object types are ranged over by the symbol σ. Nonground complex object types are obtained from ground complex object types by replacing some subexpressions with complex object type variables of the following forms:

- Unconstrained complex object type variable. It has the form `''n`, where n is a natural number. It can be instantiated to any ground complex object type.
- Record complex object type variable. It has the form `(`$l_1 : \sigma_1, \cdots, l_n : \sigma_n$`; ''m)`, where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.
- Variant complex object type variable. It has the form `<`$l_1 : \sigma_1, \cdots, l_n : \sigma_n$`; ''m>`, where m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.
- Numeric tuple object type variable. It has the form `^m>=n`, where m and n are natural numbers. It can only be instantiated to ground tuples of numbers of arity at least n.
- Subtyping object type variable. It has the form `'m<b`, where m is a natural number and b is the name of a user-defined type. It can only be instantiated to b or to its subtypes. Note that b must be a user-defined type that has no parameters. The subtyping supported by CPL allows only (explicitly declared) single inheritance.

The nonground types are ranged over by the symbol β. Nonground types are obtained from ground types by replacing some subexpressions with a universal type variable of the following forms:

- Unconstrained universal type variable. It has the form `'n`, where n is a natural number. It can be instantiated to any ground type.
- Record universal type variable. It has the form `(`$l_1 : \beta_1, \cdots, l_n : \beta_n$`; 'm)`, where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.
- Variant universal type variable. It has the form `<`$l_1 : \beta_1, \cdots, l_n : \beta_n$`; 'm>`, where m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.

Hence for example `(#age : string; '1) -> string` indicates the type of functions whose inputs are records having at least the field `#age` of `string` type and producing outputs of `string` type. Similarly, `(#age : string; '1) -> (#age : string; '1)` indicates the type of functions whose inputs are records having at least the fields `#age` of `string` type and produces outputs of exactly the same type as the inputs.

The distinction between nonground complex object types and nonground types is that the latter types include function types but the former types do not. For example, the nonground complex object type `(#input_set: {num}; ''1)` cannot be instantiated to a record type such as `(#input_set: {num}, #transformer: num->num)`. On the other hand, the nonground type `(#input_set: {num}; '1)` can be instantiated to a record type such as `(#input_set: {num}, #transformer: num->num)`.

2.2 Expressions

The expressions are ranged over by e. The variables are ranged over by x. For simplicity, we assign a ground type once-and-forever to all the variables; the type u assigned to a variable x is indicated by superscripting: x^u. Expression formation constructs are based on the structure of types.

For function types, the expression constructs are given in Figure 1. The meaning of $\backslash x^u$ `=>` e is the function f that when applied to an object o of type u produces the object $e[o/x^u]$. (The notation $e[o/x^u]$ means replace all free occurrences of x^u in e by o.) The meaning of e_1 `@` e_2 is the result of applying the function e_1 to the object e_2.

$$\frac{}{x^u : u} \qquad \frac{e : v}{\backslash x^u \texttt{ => } e : u \texttt{ -> } v} \qquad \frac{e_1 : u \texttt{ -> } v \quad e_2 : u}{e_1 \texttt{ @ } e_2 : v}$$

Figure 1: The constructs for function types in CPL.

For record types, the expression constructs are given in Figure 2. I have already mentioned that `()` is the unique object having type `unit`. The construct $(l_1 : e_1, \cdots, l_n : e_n)$ forms a record having fields l_1, ..., l_n whose values are e_1, ..., e_n respectively. A label when used as an expression stands for the obvious projection function. The construct `modify` l `of` e_1 `to` e_2 forms a new record having the same value as e_1, except at the l field which is updated to e_2.

For variant types, the expression constructs are given in Figure 3. The construct `<`$l : e$`>` forms a variant object by tagging the object e with the label l. The case-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`$l_i : o$`>`. The case-otherwise-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`$l_i : o$`>` where $1 \leq i \leq n$; otherwise it evaluates to e'.

For the base type `bool`, there are the usual constructs given in Figure 4.

For set types, the expression constructs are given in Figure 5. The meaning of $\{\}^s$ is the empty set. The meaning of $\{e\}$ is the singleton set containing e. The meaning of e_1 `{+}` e_2 is the set union of e_1 and e_2. The expression `ext`$\{e_1 \mid \backslash x^t$ `<-` $e_2\}$ stands for the set $e_1[o_1/x^t]$ `{+}` $\cdots$ `{+}` $e_1[o_n/x^t]$, where o_1, ..., o_n are all the elements of the set e_2.

$$\frac{}{\texttt{()} : \texttt{unit}} \qquad \frac{e_1 : u_1 \quad \cdots \quad e_n : u_n}{\texttt{(}l_1 : e_1, \cdots, l_n : e_n\texttt{)} : \texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)}}$$

$$\frac{}{l^{\texttt{(}l:u,\ l_1:u_1,\cdots,l_n:u_n\texttt{)}} : \texttt{(}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{)} \texttt{ -> } u}$$

$$\frac{e_1 : \texttt{(}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{)} \qquad e_2 : u}{\texttt{modify } l \texttt{ of } e_1 \texttt{ to } e_2 : \texttt{(}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{)}}$$

Figure 2: The constructs for record types in CPL.

$$\frac{e : u}{\texttt{<}l : e\texttt{>}^{\texttt{<}l_1:u_1,\cdots,l_n:u_n\texttt{>}} : \texttt{<}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{>}}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u}{\texttt{case } e \texttt{ of <}l_1 : \backslash x_1^{u_1}\texttt{> => } e_1 \texttt{ or } \cdots \texttt{ or <}l_n : \backslash x_n^{u_n}\texttt{> => } e_n : u}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_{n+m} : u_{n+m}\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u \quad e' : u}{\texttt{case } e \texttt{ of <}l_1 : \backslash x_1^{u_1}\texttt{> => } e_1 \texttt{ or } \cdots \texttt{ or <}l_n : \backslash x_n^{u_n}\texttt{> => } e_n \texttt{ otherwise } e' : u}$$

Figure 3: The constructs for variant types in CPL.

$$\frac{}{\texttt{true} : \texttt{bool}} \qquad \frac{}{\texttt{false} : \texttt{bool}} \qquad \frac{e_1 : \texttt{bool} \quad e_2 : u \quad e_3 : u}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : u}$$

Figure 4: The constructs for the Boolean type in CPL.

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \ \texttt{\{+\}}\ e_2 : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{t\}}{\texttt{ext}\{e_1 \mid \backslash x^t \texttt{ <- } e_2\} : \{s\}}$$

Figure 5: The constructs for set types in CPL.

For bag types, the expression constructs are given in Figure 6. The meaning of $\{\mid\mid\}^s$ is the empty bag. The meaning of $\{\mid e \mid\}$ is the singleton bag containing e. The meaning of $e_1\ \{\mid\texttt{+}\mid\}\ e_2$ is the bag union of e_1 and e_2; it is sometimes called the additive union. For example, if e_1 is a bag of five apples and two oranges and e_2 is a bag of one apple and three oranges, then $e_1\ \{\mid\texttt{+}\mid\}\ e_2$ is a bag of six apples and five oranges. The expression $\texttt{ext}\{\mid e_1 \mid \backslash x^t\ \texttt{<--}\ e_2 \mid\}$ stands for the bag $e_1[o_1/x^t]\ \{\mid\texttt{+}\mid\}\ \cdots\ \{\mid\texttt{+}\mid\}\ e_1[o_n/x^t]$. More information on bags can be found in [21].

$$\frac{}{\{\mid\mid\}^s : \{\mid s \mid\}} \qquad \frac{e : s}{\{\mid e \mid\} : \{\mid s \mid\}} \qquad \frac{e_1 : \{\mid s \mid\} \quad e_2 : \{\mid s \mid\}}{e_1\ \{\mid\texttt{+}\mid\}\ e_2 : \{\mid s \mid\}}$$

$$\frac{e_1 : \{\mid s \mid\} \quad e_2 : \{\mid t \mid\}}{\texttt{ext}\{\mid e_1 \mid \backslash x^t\ \texttt{<--}\ e_2 \mid\} : \{\mid s \mid\}}$$

Figure 6: The constructs for bag types in CPL.

For list types, the expression constructs are given in Figure 7. The meaning of $\texttt{[]}^s$ is the empty list. The meaning of $\texttt{[}e\texttt{]}$ is the singleton list containing e. The meaning of $e_1\ \texttt{[+]}\ e_2$ is the list concatenation of e_1 and e_2. The expression $\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{]}$ stands for the list $e_1[o_1/x^t]\ \texttt{[+]}\ \cdots\ \texttt{[+]}\ e_1[o_n/x^t]$.

$$\frac{}{\texttt{[]}^s : \texttt{[}s\texttt{]}} \qquad \frac{e : s}{\texttt{[}e\texttt{]} : \texttt{[}s\texttt{]}} \qquad \frac{e_1 : \texttt{[}s\texttt{]} \quad e_2 : \texttt{[}s\texttt{]}}{e_1\ \texttt{[+]}\ e_2 : \texttt{[}s\texttt{]}} \qquad \frac{e_1 : \texttt{[}s\texttt{]} \quad e_2 : \texttt{[}t\texttt{]}}{\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{]} : \texttt{[}s\texttt{]}}$$

Figure 7: The constructs for list types in CPL.

CPL also includes the primitives functions listed in Figure 8 for comparing complex objects. The operator `=` is the equality test. The operator `<=` is the linear order. The operator `<` is the strict version.

$$\frac{e_1 : s \quad e_2 : s}{e_1\ \texttt{=}\ e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1\ \texttt{<=}\ e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1\ \texttt{<}\ e_2 : \texttt{bool}}$$

Figure 8: The constructs for comparing objects in CPL.

CPL supports conversion between lists, bags, and sets. These operators, listed in Figure 9, correspond to the monad morphisms mentioned in Wadler [40]. The expression $\texttt{ext\{}e_1 \mid \backslash x^t\ \texttt{<--}\ e_2\texttt{\}}$ stands for the set $e_1[o_1/x^t]\ \texttt{\{+\}}\ \cdots\ \texttt{\{+\}}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the bag e_2. The expression $\texttt{ext\{|}e_1 \mid \backslash x^t\ \texttt{<-}\ e_2\texttt{|\}}$ stands for the bag $e_1[o_1/x^t]\ \texttt{\{|+|\}}\ \cdots\ \texttt{\{|+|\}}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the set e_2. The expression $\texttt{ext\{}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{\}}$ stands for the set $e_1[o_1/x^t]\ \texttt{\{+\}}\ \cdots\ \texttt{\{+\}}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the list e_2. The expression $\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<-}\ e_2\texttt{]}$ stands for the list $e_1[o_1/x^t]\ \texttt{[+]}\ \cdots\ \texttt{[+]}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the set e_2 and $o_1\ \texttt{<}\ \cdots\ \texttt{<}\ o_n$. The expression $\texttt{ext\{|}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{|\}}$ stands for the bag $e_1[o_1/x^t]\ \texttt{\{|+|\}}\ \cdots\ \texttt{\{|+|\}}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the elements in the list e_2, with o_i occurring at position i. The expression $\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<--}\ e_2\texttt{]}$ stands for the list $e_1[o_1/x^t]\ \texttt{[+]}\ \cdots\ \texttt{[+]}\ e_1[o_n/x^t]$, where o_1, ..., o_n are the elements in the bag e_2 and $o_1\ \texttt{<=}\ \cdots\ \texttt{<=}\ o_n$.

$$\frac{e_1 : \{s\} \quad e_2 : \texttt{\{|}t\texttt{|\}}}{\texttt{ext\{}e_1 \mid \backslash x^t\ \texttt{<--}\ e_2\texttt{\}} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \texttt{[}t\texttt{]}}{\texttt{ext\{}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{\}} : \{s\}}$$

$$\frac{e_1 : \texttt{\{|}s\texttt{|\}} \quad e_2 : \{t\}}{\texttt{ext\{|}e_1 \mid \backslash x^t\ \texttt{<-}\ e_2\texttt{|\}} : \texttt{\{|}s\texttt{|\}}} \qquad \frac{e_1 : \texttt{\{|}s\texttt{|\}} \quad e_2 : \texttt{[}t\texttt{]}}{\texttt{ext\{|}e_1 \mid \backslash x^t\ \texttt{<---}\ e_2\texttt{|\}} : \texttt{\{|}s\texttt{|\}}}$$

$$\frac{e_1 : \texttt{[}s\texttt{]} \quad e_2 : \{t\}}{\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<-}\ e_2\texttt{]} : \texttt{[}s\texttt{]}} \qquad \frac{e_1 : \texttt{[}s\texttt{]} \quad e_2 : \texttt{\{|}t\texttt{|\}}}{\texttt{ext[}e_1 \mid \backslash x^t\ \texttt{<--}\ e_2\texttt{]} : \texttt{[}s\texttt{]}}$$

Figure 9: The constructs for list-bag-set interactions in CPL.

CPL has a construct for sending requests to external data servers. This construct is given in Figure 10. The expression `process` e `using` N has the effect of sending the request e to the server N and the reply from N is taken as the meaning of the whole expression. See *The Kleisli Query System Reference Manual* for more information on the behaviour of servers.

$$\frac{e : s \qquad N \text{ is a server whose request type is } s \text{ and result type is } t}{\texttt{process}\ e_1\ \texttt{using}\ N : t}$$

Figure 10: The construct for sending a request to a server in CPL.

CPL supports some syntactic sugar on expressions:

- The expression $e_1(e_2)$ means $e_1\ \texttt{@}\ e_2$, provided e_1 is a function symbol.
- The expression $e_1\ \texttt{.}\ e_2$ means $e_2\ \texttt{@}\ e_1$.

- The expression $e_1\ e_2\ e_3$ is the binary function application in infix form for e_2 `@` (e_1, e_2); all binary function symbols in CPL can be applied in infix form.
- The expression e_1 `o` e_2 means $\backslash x^u$ `=>` e_1 `@` $(e_2$ `@` $x^u)$ where x^u is fresh and u is the appropriate type.
- The expression $(e_1, \cdots, e_n)$ means $(\#1 : e_1, \cdots, \#n : e_n)$.
- The expression `let` $\backslash x^s == e_1$ `in` e_2 means $(\backslash x^s$ `=>` $e_2)$ `@` e_1.
- The expression $\{e_1, \cdots, e_n\}$ means $\{e_1\}$ `{+}` $\cdots$ `{+}` $\{e_n\}$.
- The expression `{|`$e_1, \cdots, e_n$`|}` means `{|`e_1`|}` `{|+|}` $\cdots$ `{|+|}` `{|`e_n`|}`.
- The expression `[`$e_1, \cdots, e_n$`]` means `[`e_1`]` `[+]` $\cdots$ `[+]` `[`e_n`]`.
- The expression e_1 `+}` e_2 means $\{e_1\}$ `{+}` e_2.
- The expression e_1 `+|}` e_2 means `{|`e_1`|}` `{|+|}` e_2.
- The expression e_1 `+]` e_2 means `[`e_1`]` `[+]` e_2.

CPL comes with a type inference system that is considerably simpler than those of Ohori [27], Remy [30], etc., because CPL does not have a record-concatenation operation. Hence there is no need to indicate types any where in CPL expressions. So we drop our type superscripts henceforth, except when giving typing rules.

2.3 Examples: CPL's Modeling Power

Sets, lists, bags [21], records, and variants [15] are supported in CPL. These types can be freely combined, giving rise to a rich and flexible data model.

Example. Here is a list of sets of numbers in CPL:

```
[ { 1, 2, 3}, {4, 5, 6}, {}, {3, 7} ] ;
Result : [{1, 2, 3}, {4, 5, 6}, {}, {3, 7}]
Type   : [{num}]
```

Example. We can model the employee salary history example of Makinouchi [22] by a nested relation as below.

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

```
                      (#salary: 2000,
                       #date: "june 1993")},
           #name: "tom")}
Type   : {(#history:{(#salary:num, #date:string)}, #name:string)}
```

Notice that `"jim"` has the empty set as his salary history; he is probably a new employee. Had we not used nested relations, we must resort to either two flat tables (one for new employees and one for old employees) or to null values.

Example. We model student information, where some of them have phone number as contact address and some have room number instead. This is done using variants:

```
{(#name:"jim", #contact:<#phone:"3-4560">), (#name:"tom", #contact:
<#office:"2-2210">)} ;
Result : {(#contact: <#office: "2-2210">,
           #name: "tom"),
          (#contact: <#phone: "3-4560">,
           #name: "jim")}
Type : {(#contact:<#office:string,#phone:string;''2>,#name:string)}
```

Had we not used variants, we must resort to either two flat tables (one for people having phone number and one for those who have room number) or to null values.

3 Collection Comprehension in CPL

An important influence on the design of CPL is Wadler's idea of using the comprehension syntax for manipulating monads [40]. His idea is to introduce a comprehension construct $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ in place of the $\bigcup\{e_1 \mid x \in e_2\}$ construct of $\mathcal{NRC}$. This construct can be interpreted in $\mathcal{NRC}$ by treating $\{e \mid x \in e', \Delta\}$ as $\bigcup\{\{e \mid \Delta\} \mid x \in e'\}$ and $\{e \mid \}$ as $\{e\}$. Conversely the $\bigcup\{e_1 \mid x \in e_2\}$ construct can be interpreted as $\{y \mid x \in e_2,\ y \in e_1\}$ in Wadler's language. Thus his language is equivalent to $\mathcal{NRC}$.

The comprehension syntax is less abstract than $\mathcal{NRC}$ for the purpose of theoretical study. However, it is very appealing for the purpose of everyday programming. Therefore, I have added a collection comprehension mechanism to CPL. This mechanism is similar to the list comprehension mechanism in functional languages such as KRC [38] and Haskell [12]. However, CPL's version is slightly more general.

3.1 Collection Comprehensions in CPL

There are three constructs for collection comprehension, one each for sets, bags, and lists. The typing rules are given in Figure 11, where A_i and A_i* has one of the following forms:

$$\frac{e : s \quad A_1 * \quad \cdots \quad A_n *}{\{e \mid A_1, \cdots, A_n\} : \{s\}} \qquad \frac{e : s \quad A_1 * \quad \cdots \quad A_n *}{\{\mid e \mid A_1, \cdots, A_n \mid\} : \{\mid s \mid\}} \qquad \frac{e : s \quad A_1 * \quad \cdots \quad A_n *}{\texttt{[}e \mid A_1, \cdots, A_n\texttt{]} : \texttt{[}s\texttt{]}}$$

Figure 11: The comprehension constructs in CPL.

- A_i is an expression e_i. Then A_i* is the type-derivation showing e_i : `bool`.
- A_i is a set-abstraction $\backslash x^{s_i}$ `<-` e_i. Then A_i* is the type-derivation showing $e_i : \{s_i\}$.
- A_i is a bag-abstraction $\backslash x^{s_i}$ `<--` e_i. Then A_i* is the type-derivation showing $e_i : \{\mid s_i \mid\}$.
- A_i is a list-abstraction $\backslash x^{s_i}$ `<---` e_i. Then A_i* is the type-derivation showing $e_i : \texttt{[}s_i\texttt{]}$.

I now define the semantics of these comprehension constructs in terms of the various `ext` constructs introduced earlier. The translation used is based on that suggested by Wadler [40]. Let us use Δ as a meta notation for a sequence of A_i.

For set comprehensions:

- Interpret $\{e' \mid \backslash x^s\ \texttt{<-}\ e, \Delta\}$ as $\texttt{ext}\{\{e' \mid \Delta\} \mid \backslash x^s\ \texttt{<-}\ e\}$.
- Interpret $\{e' \mid \backslash x^s\ \texttt{<--}\ e, \Delta\}$ as $\texttt{ext}\{\{e' \mid \Delta\} \mid \backslash x^s\ \texttt{<--}\ e\}$.

- Interpret $\{e' \mid \backslash x^s \;\texttt{<---}\; e, \Delta\}$ as $\texttt{ext}\{\{e' \mid \Delta\} \mid \backslash x^s \;\texttt{<---}\; e\}$.
- Interpret $\{e' \mid e, \Delta\}$ as $\texttt{if}\; e \;\texttt{then}\; \{e' \mid \Delta\} \;\texttt{else}\; \{\}$.

For bag comprehensions:

- Interpret $\{\mid e' \mid \backslash x^s \;\texttt{<-}\; e, \Delta \mid\}$ as $\texttt{ext}\{\mid \{\mid e' \mid \Delta \mid\} \mid \backslash x^s \;\texttt{<-}\; e \mid\}$.
- Interpret $\{\mid e' \mid \backslash x^s \;\texttt{<--}\; e, \Delta \mid\}$ as $\texttt{ext}\{\mid \{\mid e' \mid \Delta \mid\} \mid \backslash x^s \;\texttt{<--}\; e \mid\}$.
- Interpret $\{\mid e' \mid \backslash x^s \;\texttt{<---}\; e, \Delta \mid\}$ as $\texttt{ext}\{\mid \{\mid e' \mid \Delta \mid\} \mid \backslash x^s \;\texttt{<---}\; e \mid\}$.
- Interpret $\{\mid e' \mid e, \Delta \mid\}$ as $\texttt{if}\; e \;\texttt{then}\; \{\mid e' \mid \Delta \mid\} \;\texttt{else}\; \{\mid \mid\}$.

For list comprehensions:

- Interpret $\texttt{[}e' \mid \backslash x^s \;\texttt{<-}\; e, \Delta\texttt{]}$ as $\texttt{ext[[}e' \mid \Delta\texttt{]} \mid \backslash x^s \;\texttt{<-}\; e\texttt{]}$.
- Interpret $\texttt{[}e' \mid \backslash x^s \;\texttt{<--}\; e, \Delta\texttt{]}$ as $\texttt{ext[[}e' \mid \Delta\texttt{]} \mid \backslash x^s \;\texttt{<--}\; e\texttt{]}$.
- Interpret $\texttt{[}e' \mid \backslash x^s \;\texttt{<---}\; e, \Delta\texttt{]}$ as $\texttt{ext[[}e' \mid \Delta\texttt{]} \mid \backslash x^s \;\texttt{<---}\; e\texttt{]}$.
- Interpret $\texttt{[}e' \mid e, \Delta\texttt{]}$ as $\texttt{if}\; e \;\texttt{then [}e' \mid \Delta\texttt{] else []}$.

The basic idea of interpreting comprehension in terms of the monad transformation constructs `ext` is due to Wadler [40]. Wadler explicitly considered the situation of $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ where e_1, ..., e_n come from the same monad (in this case set). He also had the idea of monad morphism that takes objects from one kind of monad to a different kind of monad. For some reason, he did not take the obvious step of building monad morphism into his comprehension syntax. My comprehension syntax directly incorporates the six special cases of monad morphism (set/bag/list-conversions) above.

3.2 Examples: Uniform Collection Manipulation with Comprehension

Comprehension notations are used in CPL to uniformly manipulate sets, lists, and bags. This mechanism is a generalization of the list comprehension mechanism in functional languages like Haskell [17], Miranda [39], KRC [38], Id [25], etc. As demonstrated by Trinder [37], this is a rather natural notation for writing queries.

Example. The cartesian product on sets can be written in CPL as below. (Note: `primitive P == e` is CPL's syntax for explicitly naming a value.)

```
primitive cpSet == (\x, \y) => { (u, v) | \u <- x, \v <- y } ;
Result : Primitive cpSet registered.
Type   : (#1: {''1}, #2: {''2}) -> {(#1: ''1, #2: ''2)}

{1,2} cpSet {3,4} ;
Result : {(#1:2, #2:4), (#1:2, #2:3), (#1:1, #2:4), (#1:1, #2:3)}
Type   : {(#1:num, #2:num)}
```

Example. The cartesian product on lists can be written in CPL as follows, where set-brackets are replaced by list-brackets and set-abstractions are replaced by list-abstractions:

```
primitive cpList == (\x, \y) => [ (u, v) | \u <--- x, \v <--- y ] ;
Result : Primitive cpList registered.
Type   : (#1: [''1], #2: [''2]) -> [(#1: ''1, #2: ''2)]

["a", "b"] cpList ["c", "d"] ;
Result : [(#1: "a", #2: "c"), (#1: "a", #2: "d"),
          (#1: "b", #2: "c"), (#1: "b", #2: "d")]
Type   : [(#1:string, #2:string)]
```

Example. Conversion between lists, bags, and sets is very natural. Here is the function that selects all positive numbers in a list and puts them in a set. (Note that CPL uses `~1`, `~3`, etc. for negative numbers.)

```
primitive positiveListToSet == \x => { y | \y <--- x, 0 <= y } ;
Result : Primitive positiveListToSet registered.
Type   : [num]->{num}

positiveListToSet @ [~1, 2, ~3, 5] ;
Result : { 2, 5 }
Type   : {num}
```

4 Pattern Matching in CPL

To further increase the user-friendliness of CPL queries, I add a pattern-matching mechanism to CPL. This mechanism is more general than that found in languages such as HOPE [7] and ML [23]. In particular, it supports partial-record patterns found in Machiavelli [26] and it supports variable-as-constant patterns not found anywhere else. I introduce the pattern matching mechanism in two stages, viz. simple patterns and enhanced patterns.

4.1 Simple Patterns

I use the meta symbol S to range over simple patterns. The grammar is given below:

S	$::=$	$\_$	Match anything
	$\mid$	$\backslash x$	Match anything and bind it to x
	$\mid$	$\backslash x\texttt{\&}S$	Match using S and bind it to x
	$\mid$	$\texttt{(}l_1 : S_1, \cdots, l_n : S_n\texttt{)}$	Match records
	$\mid$	$\texttt{(}l_1 : S_1, \cdots, l_n : S_n, ...\texttt{)}$	Match records partially
	$\mid$	`()`	Match `()` only

A pattern must also satisfy the constraint that no $\backslash x$ is allowed to appear more than once in it. The last three dots in the pattern $\texttt{(}l_1 : S_1, \cdots, l_n : S_n, ...\texttt{)}$ are part of the syntax; this is called the partial record patten. Also I say a pattern is ultra-simple if it is just $\backslash x$. CPL also support the pattern $\texttt{(}S_1, \cdots, S_n\texttt{)}$ as syntactic sugar for the pattern $\texttt{(}\#1 : S_1, \cdots, \#n : S_n\texttt{)}$.

Simple patterns are used in lambda abstraction, case-expression, case-otherwise-expression, set abstraction, bag abstraction, and list abstraction. That is, they can be used anywhere a $\backslash x$ can be used. I now define the semantics of these patterns in terms of the core language presented earlier. The translation is given by cases below.

For lambda abstraction:

- Treat $\_\ \texttt{=>}\ e$ as $\backslash x\ \texttt{=>}\ e$ where x is fresh.
- Treat $\backslash x\texttt{\&}S\ \texttt{=>}\ e$ as $\backslash x\ \texttt{=>}\ (S\ \texttt{=>}\ e)\ \texttt{@}\ x$.
- Treat $\texttt{(}l_1 : S_1, \cdots, l_n : S_n\texttt{)}\ \texttt{=>}\ e$ as $\backslash x\ \texttt{=>}\ (S_1\ \texttt{=>}\ \cdots (S_n\ \texttt{=>}\ e\ \texttt{@}\ (x\ \texttt{.}\ l_n)) \cdots)\ \texttt{@}\ (x\ \texttt{.}\ l_1)$
- Treat $\texttt{(}l_1 : S_1, \cdots, l_n : S_n, ...\texttt{)}\ \texttt{=>}\ e$ as $\backslash x\ \texttt{=>}\ (S_1\ \texttt{=>}\ \cdots (S_n\ \texttt{=>}\ e\ \texttt{@}\ (x\ \texttt{.}\ l_n)) \cdots)\ \texttt{@}\ (x\ \texttt{.}\ l_1)$
- Treat $\texttt{()}\ \texttt{=>}\ e$ as $\backslash x^{\texttt{unit}}\ \texttt{=>}\ e$, where $x^{\texttt{unit}}$ is fresh.

For case-expressions:

- Treat $\texttt{case}\ e\ \texttt{of}\ \texttt{<}l_1 : S_1\texttt{>}\ \texttt{=>}\ e_1\ \texttt{or}\ \cdots\ \texttt{or}\ \texttt{<}l_n : S_n\texttt{>}\ \texttt{=>}\ e_n$ as $\texttt{case}\ e\ \texttt{of}\ \texttt{<}l_1 : \backslash x_1\texttt{>}\ \texttt{=>}\ (S_1\ \texttt{=>}\ e_1)\ \texttt{@}\ x_1\ \texttt{or}\ \cdots\ \texttt{or}\ \texttt{<}l_n : \backslash x_n\texttt{>}\ \texttt{=>}\ (S_n\ \texttt{=>}\ e_n)\ \texttt{@}\ x_n$, where all x_i are fresh and some S_i are not ultra-simple.

- Treat $\texttt{case}\ e\ \texttt{of <}l_1\ \texttt{:}\ S_1\texttt{> =>}\ e_1\ \texttt{or}\ \cdots\ \texttt{or <}l_n\ \texttt{:}\ S_n\texttt{> =>}\ e_n\ \texttt{otherwise}\ e'$ as $\texttt{case}\ \ e\ \ \texttt{of <}\ l_1\ \ \texttt{:}\ \backslash x_1\ \texttt{>}\ \ \texttt{=>}\ (S_1\ \ \texttt{=>}\ e_1)\ \texttt{@}\ x_1\ \texttt{or}\ \cdots\ \texttt{or <}l_n : \backslash x_n\texttt{> =>}\ (S_n\ \ \texttt{=>}\ e_n)\ \texttt{@}\ x_n\ \ \texttt{otherwise}\ e'$, where all x_i are fresh and some S_i are not ultra-simple.

For collection abstractions, I provide only the cases of `ext` for illustration. The cases for `ext` and `ext` are analogous.

- Treat $\texttt{ext}\{e_1 \mid S\ \texttt{<-}\ e_2\}$ as $\texttt{ext}\{(S\ \texttt{=>}\ e_1)\ \texttt{@}\ x \mid \backslash x\ \texttt{<-}\ e_2\}$, where x is fresh and S is not ultra-simple.
- Treat $\texttt{ext}\{e_1 \mid S\ \texttt{<--}\ e_2\}$ as $\texttt{ext}\{(S\ \texttt{=>}\ e_1)\ \texttt{@}\ x \mid \backslash x\ \texttt{<--}\ e_2\}$, where x is fresh and S is not ultra-simple.
- Treat $\texttt{ext}\{e_1 \mid S\ \texttt{<---}\ e_2\}$ as $\texttt{ext}\{(S\ \texttt{=>}\ e_1)\ \texttt{@}\ x \mid \backslash x\ \texttt{<---}\ e_2\}$, where x is fresh and S is not ultra-simple.

4.2 Enhanced Patterns

I use the meta symbol E to range over enhanced patterns. Enhanced patterns are a generalization of simple patterns. The grammar is given below:

E	$::=$	$\_$	Match anything
	$\mid$	$\backslash x$	Match anything and bind it to x
	$\mid$	$\backslash x\texttt{\&}E$	Match using E and bind it to x
	$\mid$	$(l_1 : E_1, \cdots, l_n : E_n)$	Match records
	$\mid$	$(l_1 : E_1, \cdots, l_n : E_n, ...)$	Match records partially
	$\mid$	`()`	Match `()` only
	$\mid$	c	Match constant c only
	$\mid$	$\texttt{<}l : E\texttt{>}$	Match variants
	$\mid$	x	Match the value bound to x only

Observe that in simple patterns every occurrence of a variable x is slashed, as in $\backslash x$. In enhanced patterns, a variable can appear without being slashed. A pattern where a variable x occurs without being slashed is called a variable-as-constant pattern. As before, a pattern must satisfy the constraint that no $\backslash x$ is allowed to appear more than once in it; however, unslashed variables can appear as frequently as desired.

Enhanced patterns are used only in set abstraction, bag abstraction, and list abstraction. I define their semantics in terms of simple patterns. I give the cases for `ext` for illustrations. The other cases are analogous.

- Treat $\texttt{ext}\{e_1 \mid E\ \texttt{<-}\ e_2\}$ as $\texttt{ext}\{\texttt{if}\ x = A\ \texttt{then}\ e_1\ \texttt{else}\ \{\} \mid E'\ \texttt{<-}\ e_2\}$, where x is fresh, A is a subpattern in E and is either a constant or an unslashed variable, and E' is obtained from E by replacing one occurrence of A with $\backslash x$.

- Treat $\texttt{ext}\{e_1 \mid E \texttt{ <- } e_2\}$ as $\texttt{ext}\{\texttt{case } x \texttt{ of <}l : \backslash y\texttt{> => ext}\{e_1 \mid E'' \texttt{ <- } \{y\}\} \texttt{ otherwise } \{\} \mid E' \texttt{ <- } e_2\}$, where x and y are fresh, $\texttt{<}l : E''\texttt{>}$ is a subpattern in E and E' is obtained from E by replacing one occurrence of $\texttt{<}l : E''\texttt{>}$ by $\backslash x$.

- Treat $\texttt{ext}\{e_1 \mid E \texttt{ <-- } e_2\}$ as $\texttt{ext}\{\texttt{if } x = A \texttt{ then } e_1 \texttt{ else } \{\} \mid E' \texttt{ <-- } e_2\}$, where x is fresh, A is a subpattern in E and is either a constant or an unslashed variable, and E' is obtained from E by replacing one occurrence of A with $\backslash x$.

- Treat $\texttt{ext}\{e_1 \mid E \texttt{ <-- } e_2\}$ as $\texttt{ext}\{\texttt{case } x \texttt{ of <}l : \backslash y\texttt{> => ext}\{e_1 \mid E'' \texttt{ <- } \{y\}\} \texttt{ otherwise } \{\} \mid E' \texttt{ <-- } e_2\}$, where x and y are fresh, $\texttt{<}l : E''\texttt{>}$ is a subpattern in E and E' is obtained from E by replacing one occurrence of $\texttt{<}l : E''\texttt{>}$ by $\backslash x$.

- Treat $\texttt{ext}\{e_1 \mid E \texttt{ <--- } e_2\}$ as $\texttt{ext}\{\texttt{if } x = A \texttt{ then } e_1 \texttt{ else } \{\} \mid E' \texttt{ <--- } e_2\}$, where x is fresh, A is a subpattern in E and is either a constant or an unslashed variable and E' is obtained from E by replacing one occurrence of A with $\backslash x$.

- Treat $\texttt{ext}\{e_1 \mid E \texttt{ <--- } e_2\}$ as $\texttt{ext}\{\texttt{case } x \texttt{ of <}l : \backslash y\texttt{> => ext}\{e_1 \mid E'' \texttt{ <- } \{y\}\} \texttt{ otherwise } \{\} \mid E' \texttt{ <--- } e_2\}$, where x and y are fresh, $\texttt{<}l : E''\texttt{>}$ is a subpattern in E and E' is obtained from E by replacing one occurrence of $\texttt{<}l : E''\texttt{>}$ by $\backslash x$.

This is a good place to explain the motivation of slashing a variable on its introduction. Consider the expression $\lambda x.\{(x,y) \mid (x,y) \in R\}$ written in a comprehension notation consistent with Wadler's [40]. On first sight, this seems to be a program which takes in an x and then selects from the relation R every pair whose first component is equal to this x. However, this obvious impression is incorrect. The expression above is equivalent to $\lambda x.\{(x',y) \mid (x',y) \in R\}$ with x' a fresh variable. That is, it takes in an x and reproduces an exact copy of the relation R.

Variable-slashing reduces this kind of mistake because it makes the above expression illegal. To see this, let me rewrite the expression in CPL without inserting the proper slashes: $\backslash x \texttt{ => } \{\texttt{(}x,y\texttt{)} \mid \texttt{(}x,y\texttt{)} \texttt{ <- } R\}$. Now this expression is no longer closed because y has become a free variable. The CPL program that implements the obvious but incorrect meaning of the original expression is: $\backslash x \texttt{ => } \{\texttt{(}x,y\texttt{)} \mid \texttt{(}x,\backslash y\texttt{)} \texttt{ <- } R\}$. The CPL program that implements the correct but obscured meaning of the original expression is: $\backslash x \texttt{ => } \{\texttt{(}x,y\texttt{)} \mid \texttt{(}\backslash x,\backslash y\texttt{)} \texttt{ <- } R\}$. The absence and presence of the slash in front of the third x makes the difference very clear.

4.3 Examples: Convenience of Pattern Matching

It is generally agreed that pattern matching makes queries more readable. Here are some examples to illustrate CPL's pattern-matching mechanism.

Example. To illustrate partial-record patterns, here is a CPL query for finding the names of children who are ten years old:

```
primitive ten_year_olds ==
  \people => { x | (#name: \x, #age: 10, ...) <- people} ;
Result : Primitive ten_year_olds installed.
```

```
Type   : {(#name : ''1, #age: num; ''2)}->{''1}

ten_year_olds @ {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")};
Result : {"tom"}
Type   : {string}
```

Example. To illustrate layered patterns, here is the CPL query that returns those children who are ten years old (that is, not just their names):

```
primitive ten_year_olds' ==
  \people => { y | \y&(#name: \x, #age: 10, ...) <- people} ;
Result : Primitive ten_year_olds' installed.
Type   : {(#name:''1, #age:num; ''2)}->{(#name:''1, #age:num;''2)}

ten_year_olds' @ {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")};
Result : {(#name : "tom", #age : 10, #sex : "male")}
Type   : {(#name : string, #age : num, #sex : string)}
```

Example. To illustrate variable-as-constant patterns, consider generalizing `ten_year_olds` to find names of children who are x years old, where x is to be given. Here is the query in CPL:

```
primitive x_year_olds ==
  (\people, \x) => { y | (#name: \y, #age: x, ...) <- people } ;
Result : Primitive x_year_olds installed.
Type   : (#1: {(#name : ''1, #age: ''2; ''3)}, #2: ''2)->{''1}

x_year_olds @ ( {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")}, 12) ;
Result : {"jim"}
Type   : {string}
```

Notice that the `10` in the `ten_year_olds` query is simply replaced by `x`, the input to be given. Since this occurrence of x is not slashed, it is not the introduction of a new variable. Rather it stands for the value that is supplied to the function as its second argument (that is, the `\x` argument). This kind of pattern is not found in any other pattern-matching language with which I am acquainted. (Prolog [33] does support a pattern mechanism based on unification which can be used to simulate my variable-as-constant patterns. However, Prolog is not a pattern-matching language. The task of matching Q against a pattern P is in finding a substitution θ so that $Q = (P)\theta$. The task of unifying Q and P is in finding a substitution θ so that $(Q)\theta = (P)\theta$. The two are clearly different.)

Without the variable-as-constant pattern mechanism, the same function would have to be written using an explicit equality test, producing a query that is quite different from `ten_year_olds`:

```
primitive x_year_olds' ==
  (\people,\x) => {y | (#name:\y, #age:\z, ...) <- people, z = x};
Result : Primitive x_year_olds' installed.
Type   : (#1: {(#name : ''1, #age: ''2; ''3)}, #2: ''2)->{''1}
```

Example. As the final pattern-matching example, here is a CPL query that computes the average salary of employees in departments. (This example uses a primitive `average` to compute the average of a bag of numbers. It can be defined either using the standard collection of primitives in CPL given in the Appendix or as an external function. See Section 5 for a description of how to add an external function.)

```
primitive ave_sal_by_dept == \DB =>
  {(#dept: x,
    #ave_sal: average @ {| y | (#dept:x, #sal:\y, ...) <- DB |})
  | (#dept: \x, ...) <- DB};
Result : Primitive ave_sal_by_dept installed.
Type : {(#sal:num, #dept:''10; ''3)}->{(#ave_sal:num, #dept:''10)}


ave_sal_by_dept @ {
  (#dept:  "cis", #emp: "john", #sal: 1000),
  (#dept:  "cis", #emp: "jeff", #sal: 1000),
  (#dept:  "cis", #emp: "jack", #sal:  400),
  (#dept: "math", #emp: "jane", #sal:  900),
  (#dept: "math", #emp: "jill", #sal:  600),
  (#dept:  "phy", #emp: "jean", #sal: 2000)};
Result : {(#ave_sal: 2000.0, #dept: "phy"),
          (#ave_sal: 750.0, #dept: "math"),
          (#ave_sal: 800.0, #dept: "cis")}
Type   : {(#ave_sal:num, #dept:string)}
```

Note the conversion to bag in the query, which captures the semantics of *group-by* in SQL. Without this conversion, then John and Jeff in the example input will cause the average salary in the CIS Department to be miscounted.

5 Other Features of CPL

I have mentioned a few other features of CPL earlier on: it has a type inference system, it is extensible, and it has an optimizer. I use some simple examples to illustrate them here.

5.1 Types Are Automatically Inferred

The type system is simpler than that of Ohori [27], Remy [30], and Jategaonkar and Mitchell [19]. The reason for this is that CPL does not have a record concatenation mechanism. For example, CPL infers that `ten_year_olds` has unique most general type `{(#name : ''1, #age: num; ''3)}->{''1}`.

5.2 Easy to Add New Primitives

The core of CPL is not a very expressive language. In fact, when restricted to sets, it is equivalent to the well-known nested relational algebra of Thomas and Fischer [35]. Therefore, it has to be augmented with extra primitives that reflect the needs of the applications that non-expert users are trying to solve. The extra primitives are provided by expert users who build them in the host language. Non-expert users only need to import them into CPL.

It is very easy to extend CPL with new primitives. I illustrate this feature by showing how to insert a factorial function into CPL. The expert user first programs the factorial function `hostFact` in his host language, and registers it with the Kleisli query system as `factorial` by a simple library call in his host language. The host language is ML [23], it is not to be confused with CPL. I use `! beginsml` and `! endsml` to mark ML programs to avoid confusion. (In ML, `F o G` stands for composition of functions `F` and `G` and `M.F` stands for the function `F` in the module `M`.)

```
! beginsml
(* The SML implementation of "hostFact" *)
    fun hostFact n = if n < 1 then 1 else n * (hostFact (n - 1));

(* Register it for use inside CPL as "factorial" *)
    Primitive.Add
      "factorial"
      (COFunction.Mk(CONum.Mk o hostFact o CONum.Km))
      (TYI.ReadFromString "num -> num");
! endsml
```

`Primitive.Add` is the registration routine. `TYI.ReadFromString` is the routine for converting a type specification given in a string to the internal format used by Kleisli. `COFun.Mk`, `CONum.Mk`, and `CONum.Km` are routines for converting between the Kleisli's and the host language's representations of complex objects. These routines are provided in the libraries of the Kleisli query system.

The non-expert user can then begin using the new primitive `factorial`.

```
factorial @ 5 ;
Result : 120
Type   : num
```

5.3 Easy to Add New Data Writers

To be useful a query language must be able to produce external data. CPL uses “writers” for writing external data in various format. It is easy to add new writers to CPL. A writer is a special kind of data driver that accepts a triple as its input. The first component of the input is an object representing the output destination, which is usually just a file name. The second component of the input is required to be a string, which is assumed represent the type of the output object. Finally, the third component is the output object itself.

Below is an example program which adds the standard writer `stdout` to CPL. The ML function `Server.Add` is the registration routine. The ML function `stdoutSv` is our implementation of the standard writer. See *The Kleisli Query System Reference Manual* for more information on `Server.Add`.

```
! beginsml
    Server.Add
       "stdout"
       stdoutSv
       (TYI.ReadFromString "(string, string, ''1)") (* filename, type, output *)
       (fn _ => TYI.ReadFromString "string")        (* reply is string *)
       1                                            (* 1 server is enough. *)
! endsml
```

A non-expert user can use the `writefile` DATA `to` SINK `using` WRITER command for producing external data, as in the CPL example below.

```
writefile {1,2,3} to "temp" using stdout;
Result: File temp written.
Type: {num}
```

Recall that a writer is just a special kind of data server. Hence it can also be invoked using standard server commands. For example, the above is same as `process ("temp", "{num}", {1,2,3}) using stdout`.

5.4 Easy to Add New Scanners

To be useful a query language must be able to read external data. CPL uses “scanners” for reading external data in various format. It is easy to add new scanners to CPL. A scanner is just a special kind of data driver. There are two things associated with a scanner. The first is its input, which specifies where to get the data from. The second is a function for computing the type of the data read, given the input specification.

Here is a program that adds the standard scanner `stdin` to CPL. The ML function The ML function `stdinSv` is our implementation of the standard input server. See *The Kleisli Query System Reference Manual* for details.

```
;
! beginsml
   let fun GetSchema (MediumRare.String F) =
         if System.Unsafe.SysIO.access (F ^ ".typ", [])
         then TYI.ReadFromFile (F ^ ".typ")
         else Error.ReportFatal ("STDIO.stdinSv: File "^F^".typ not found.")
       | GetSchema _ = TYI.ReadFromString "'1"
   in Server.Add
         "stdin"
         stdinSv
         (TYI.ReadFromString "string")        (* name of input file. *)
         GetSchema                            (* get schema of input file. *)
         1                                    (* 1 server is enough. *)
   end
! endsml
;
```

After that, a non-expert user can read external data files in Kleisli's standard exchange format using the `readfile` NAME `from` SOURCE `using` SCANNER command. For example, the file `temp` written out earlier can now be read in.

```
readfile pmet from "temp" using stdin;
Result: File pmet registered.
Type: {num}

pmet;
Result: {1, 2, 3}
Type: {num}
```

Recall that a scanner is just a special kind of data server. So it can also be invoked using standard server commands. For example, the above is same as `primitive pmet == process "temp" using stdin`.

5.5 An Extensible Optimizer is Available

CPL is equipped with an extensible optimizer. The optimizer does pipelining, joins, caching, and many other kinds of optimization. I illustrate it here on a very simple query. First, let me create a text file.

```
writefile {{(1,2), (3,4)}, {(5, 6), (7, 8)}} to "tmp" using stdout;
Result : File tmp written.
Type   : {{(#1:num, #2:num)}}
```

Now we query the file by doing a flatten and a projection operation on it:

```
readfile db from "tmp" using stdin;
Result : File db registered.
Type   : {{(#2:num, #1:num)}}

{ x | \X <- db, (\x, _) <- X } ;
Result : {1, 3, 5, 7}
Type   : {num}
```

Without the optimizer, the peak space requirement is memory to hold 4 integers and nothing gets printed until the entire set $\{1, 3, 5, 7\}$ has been constructed. With the optimizer, the peak space requirement for this query is space for 1 integer and the first element of the output is printed instantly (while the rest of the output is still being computed). The reason is that the optimizer is sophisticated enough to push the projection on `(\x, _)` and the printing of `x` directly into the scanning of the input file `"tmp"`. (Note that the `readfile db from "tmp"` part of the query does not actually read the file; it merely establishes the file `"tmp"` as an input stream.)

6 Subtyping in CPL

CPL supports a limited form of subtyping. Subtyping is restricted to user-defined types that have no parameters. these types can be thought of as abstract data types. I first give an idea of how to declare user-defined types and assert subtype relationship between them. Then I give examples to illustrate the use of subtyping in CPL queries.

6.1 User-Defined Types

One can extend CPL with more "base" types by simplying registering new user-defined types in Kleisli. I will illustrate how this is done using a simple example. Consult *The Kleisli Query System Reference Manual* for more detail.

First let us add a couple of "classes" called `people` and `employee`. I have also declared `employee` to be a subtype of `people`. As mentioned before, CPL supports only single inheritance. Also immediate subtype-supertype relationship has to be explictly registered. In addition, at the point when a type X is registered as a subtype of a type Y, the type X must not currently be a supertype of any type. After such a relationship is registered, one can add a third type Z and make it a subtype of X. The system will automatically infer that Z is also a subtype of Y.

```
;
! beginsml
(* Add a "people" and a "employee" class to CPL. *)
val peopleTNo = COBase.Add "people" NONE NONE NONE;
val employeeTNo = COBase.Add "employee" NONE NONE NONE;
(* Make "employee" subtype of "people". *)
val _ = TY.AddSubTy employeeTNo peopleTNo;
! endsml
;
```

After the new types above have been registered, we can add some functions to create and manipulate objects of these types. Let us first add a function `people-mk` for creating a `people` object from a string standing for his name and a set of strings standing for things that he likes. We also add functions `people-name` and `people-likes` that given a `people` object return his name and the things he likes respectively. Notice that this process is very similar to implementing an abstract data type!

```
;
! beginsml
(* Add function for making people. *)
val peopleMkPNo = Primitive.Add
  "people-mk"
  (COFun.Mk (COBase.Mk peopleTNo))
  (TYI.ReadFromString "(string, {string}) -> people");
```

```
(* Add functions for accessing attributes of people. *)
val peopleNamePNo = Primitive.Add
  "people-name"
  (COFun.Mk (CORecord.Proj1 o COBase.Km))
  (TYI.ReadFromString "'1<people -> string");
val peopleLikesPNo = Primitive.Add
  "people-likes"
  (COFun.Mk (CORecord.Proj2 o COBase.Km))
  (TYI.ReadFromString "'1<people -> {string}");
! endsml
;
```

Example. Let us see the types of these functions and create some `people` for later use:

```
people-mk;
Result: --- function---
Type: (#1: string, #2: { string }) -> people

people-name;
Result: --- function---
Type: '1<people -> string

people-likes;
Result: --- function---
Type: '1<people -> { string }

primitive People == {
  people-mk ("tom", {"math", "physics", "apples"});
  people-mk ("peter", {"database", "wine", "scotland"}),
  people-mk ("anthony", {"teddy bear", "apples"}) };
Result: Primitive People registered.
Type: { people }

People;
{people (#1: "anthony", #2: {"teddy bear", "apples"}),
 people (#1: "peter", #2: {"database", "wine", "scotland"}),
 people (#1: "tom", #2: {"math", "physics", "apples"})}
Type: { people }
```

Now let us add a function `employee-mk` to create an `employee` object from a string (his name), a set of strings (things he likes), and a number (his salary). We also add a function `employee-salary` to extract the salary of an `employee`. But we do not need to add functions for extracting his name and the things he like, because we can use `people-name` and `people-likes` instead.

```
;
! beginsml
val employeeMkPNo = Primitive.Add
  "employee-mk"
  (COFun.Mk (COBase.Mk employeeTNo))
  (TYI.ReadFromString "(string, {string}, num) -> employee");
val employeeSalaryPNo = Primitive.Add
  "employee-salary"
  (COFun.Mk (CORecord.Proj3 o COBase.Km))
  (TYI.ReadFromString "'1<employee -> num");
! endsml
;
```

Example. Let us see the type of these functions and create some `employees` for later use.

```
employee-mk;
Result: --- function---
Type: (#1: string, #2: { string }, #3: num) -> employee

employee-salary;
Result: --- function---
Type: '1<employee -> num

primitive John == employee-mk ("john", {"beer", "scotland"}, 1000);
Result: Primitive John registered.
Type: employee

John;
Result: employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000)
Type: employee

primitive Employees == { John,
  employee-mk ("steve", {"apples", "oranges", "cars"}, 2000) };
Result: Primitive Employees registered.
Type: { employee }

Employees;
Result: {employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000),
         employee (#1: "steve", #2: {"apples", "oranges", "cars"}, #3: 2000)}
Type: { employee }
```

6.2 Type Casting

To make things more interesting, let us also have a function to do type casting between `employees` and `people`. We give an implementation that is interesting: we implement the type casting function as a data driver!

```
;
! beginsml
val peoplecastPNo = Server.Add
  "people-cast"
  (let fun shutdown () = ()
       fun dispatch Req RepCh ReadyCh =
          let val (CO.String S, C as CO.User (TNo, _)) = CORecord.KmPair Req
              val Rep = (case COBase.PeekTNo S
                 of SOME TNo' => if TY.IsSubTyOf TNo TNo'
                      then COSet.Singleton C
                      else COSet.Empty
                 | NONE => COSet.Empty)
          in CML.send (RepCh, Rep);
             CML.send (ReadyCh, ())
          end
    in fn () => (dispatch, shutdown) end)
  (TYI.ReadFromString "(string, '1<people)")
  (fn R => case R
       of MediumRare.Record X => (case Record.Proj1 X
           of MediumRare.String S => (case COBase.PeekTNo S
               of SOME TNo => TYI.ReadFromString ("{" ^ S ^ "}")
               | NONE => Error.ReportFatal "non-existent type.")
           | _ => TYI.ReadFromString "{ '1<people}")
       | _ => TYI.ReadFromString "{ '1<people}")
  1;
! endsml
;
```

The last few lines above, `(fn R => case R ...)`, is perhaps the most interesting part of the implementation. It implements a function that will find out the type of the output that will be produced by the data driver `people-cast` given an input R. Consult *The Kleisli Query System Reference Manual* for details.

Example. To see what is so interesting about the above implementation, let us see the types of the results below. Indeed, the type of `people-cast(`S`, `X`)` depends on what is the value of the string S! This illustrate a feature of the way CPL's type system interface with its data drivers: The CPL type inference system actually asks the “data driver” `people-cast` what should the type of (S, X) be.

```
people-cast;
```

```
Result: --- function---
Type: (#1: string, #2: '4<people) -> { '3<people }

\x => people-cast ("employee", x);
Result: --- function---
Type: '4<people -> { employee }

\x => people-cast ("people", x);
Result: --- function---
Type: '4<people -> { people }

people-cast ("employee", John);
Result: {employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000)}
Type: { employee }

people-cast ("people", John);
Result: {employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000)}
Type: { people }
```

6.3 Examples: Uses of Subtyping

Example. We are now ready to demonstrate the finer aspects of subtyping in CPL. The type system still insists on homogeneous collections. So if you want to form a set of people and employees, you will need to cast employees into people.

```
primitive All == People {+}
  { y | \x <- Employees, \y <- people-cast ("people", x) };
Result: Primitive All registered.
Type: { people }

All;
Result: {people (#1: "anthony", #2: {"teddy bear", "apples"}),
         people (#1: "peter", #2: {"database", "wine", "scotland"}),
         people (#1: "tom", #2: {"math", "physics", "apples"}),
         employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000),
         employee (#1: "steve", #2: {"apples", "oranges", "cars"}, #3: 2000)}
Type: { people }
```

Example. Even though the set `All` now contains both people and employees, you can use `people-name` to find out all their names. That function works on all subtypes of `people`. Just in case you are not convinced, we extract the name of employee `John` too.

```
{ people-name (x) | \x <- All };
```

```
Result: {"anthony", "peter", "tom", "john", "steve"}
Type: { string }

people-name (John);
Result: "john"
Type: string
```

Example. It is also possible to declare functions that work on multiple subtypes at the same time. Let us define a function to test if two people are compatible — they are if they share some common interests. Then we use it to find people compatible with `John` and also to group people by their compatibility.

```
primitive compatible == (\x, \y) =>
  not (((people-likes (x)) set-intersect (people-likes (y))) = {});
Result: Primitive compatible registered.
Type: (#1: '11<people, #2: '14<people) -> bool

{ people-name (x) | \x <- All, compatible (x, John) };
Result: {"peter", "john"}
Type: { string }

{ { people-name (y) | \y <- All, compatible (x, y) } | \x <- All };
Result: {{"anthony", "tom", "steve"}, {"peter", "john"}}
Type: { { string } }
```

Example. Let us extract from the set of people `All` those employees who earn more than 2000 dollars. Note the use of `people-cast` to test if an object x in `All` is an employee.

```
  { y | \x <- All,
        \y <- people-cast ("employee", x),
       1500 < employee-salary (y) };
Result: {employee (#1: "steve", #2: {"apples", "oranges", "cars"}, #3: 2000)}
Type: { employee }
```

Example. Unlike some subtyping systems, CPL does not suffer from the problem of "loss of type information." Suppose we want to have a function `is-john` that given any object X of type `people` or its subtype, tests if its name is "john" and if so returns a singleton set containing X otherwise returns the empty set. It is desirable that the output set should have type set of s, where s is the type of X. A function having this property is impossible to implement in "lossy" system such as that of C++. But it is simple to implement in CPL.

```
primitive is-john == \x => { x | people-name(x) = "john" };
Result: Primitive is-john registered.
```

```
Type: '5<people -> { '5<people }

is-john (John);
Result: {employee (#1: "john", #2: {"beer", "scotland"}, #3: 1000)}
Type: { employee }
```

References

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

[2] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[3] P. Bucher, K. Karplus, K. Moeri, and K. Hoffman. A flexible search technique based on generalized profiles. *Computers and Chemistry*, 20:3–24, 1996.

[4] 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.

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

[6] 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.

[7] R. M. Burstall, D. B. Macqueen, and D. T. Sanella. HOPE: An experimental applicative language. In *Proceedings of 1st LISP Conference*, pages 136–143, Stanford, California, 1980.

[8] Richard Durbin and Jean Thierry Mieg. A C. elegans Database, 1991. Documentation, code, and data available from FTP servers at lirmm.lirmm.fr, cele.mrc-lmb.cam.ac.uk and ncbi.nlm.nih.gov.

[9] S. R. Eddy. Multiple alignment using hidden Markov models. *Intelligent Systems for Molecular Biology*, 3:114–120, 1995.

[10] S.R. Eddy. Hidden Markov models. *Current Opinion in Structural Biology*, 6:361–365, 1996. HMM in protein.

[11] S.R. Eddy, G. Mitchison, and R. Durbin. Maximum discrimination hidden Markov models of sequence consensus. *Journal of Computational Biology*, 2(9–23), 1995.

[12] Joseph H. Fasel, Paul Hudak, Simon Peyton-Jones, and Philip Wadler. The functional programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[13] K. Fukuda, A. Tamura, T. Tsunoda, and T. Takagi. Toward information extraction: Identifying protein names from biological papers. In *Proceedings of Pacific Symposium on Biocomputing'98*, pages 707–718, Maui, Hawaii, January 1998.

[14] Pijush K. Ghosh. A mathematical model for shape description using Minkowski operators. *Computer Vision, Graphics, and Image Processing*, 44:239–269, 1988.

[15] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[16] Paul Horton and Kenta Nakai. Better prediction of protein cellular localization sites with the k nearest neighbours classifier. *Intelligent Systems for Molecular Biology*, 5:147–152, 1997.

[17] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report 90/?, Glasgow University, Glasgow G12 8QQ, Scotland, April 1990.

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

[19] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[20] S. Karlin and S. F. Altschul. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proc. Natl. Acad. Sci. USA*, 90:5873–5877, June 1993.

[21] Leonid Libkin and Limsoon Wong. Some properties of query languages for bags. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 97–114. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[22] Akifumi Makinouchi. A consideration on normal form of not necessarily normalised relation in the relational data model. In *Proceedings of 3rd International Conference on Very Large Databases, Tokyo, Japan*, pages 447–453, October 1977.

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

[24] A. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. SCOP: A structural classification of protein database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540, 1995.

[25] R. S. Nikhil. The parallel programming language Id and its compilation for parallel machines. In *Proceedings of Workshop on Massive Parallelism, Amalfi, Italy, October 1989.* Academic Press, 1990.

[26] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[27] Atsushi Ohori. *A Study of Semantics, Types, and Languages for Databases and Object Oriented Programming.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 1989.

[28] Anders Gorm Pedersen and Henrik Nielsen. Neural network prediction of translation initiation sites in eukaryotes: Perspectives for EST and genome analysis. *Intelligent Systems for Molecular Biology*, 5:226–233, 1997.

[29] M. Petre and R. L. Winder. Issues governing the suitability of programming languages for programming tasks. In *People and Computers IV: Proceedings of HCI'88*, Cambridge, 1988. Cambridge University Press.

[30] Didier Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[31] P. M. Sharp and W. H. Li. The codon adaptation index–a measure of directional synonymous codon usage bias, and its potential applications. *Nucleic Acids Research*, 15(3):1281–1295, February 1987.

[32] E.L. Sonnhammer, S.R. Eddy, and R. Durbin. Pfam: A comprehensive database of protein families based on seed alignments. *Proteins*, 28:405–420, 1997.

[33] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[34] Dan Suciu and Limsoon Wong. On two forms of structural recursion. In *LNCS 893: Proceedings of 5th International Conference on Database Theory*, pages 111–124, Prague, January 1995. Springer-Verlag.

[35] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis and F. P. Preparata, editors, *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

[36] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties, and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[37] 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.

[38] David Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and David Turner, editors, *Functional Programming and its Applications.* Cambridge University Press, 1982.

[39] David Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *LNCS 201: Proceedings of Conference on Functional Programming Languages and Computer Architecture, Nancy, 1985*, pages 1–16. Springer-Verlag, 1985.

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

[41] Limsoon Wong. *Querying Nested Collections.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

[42] John Wootton and Scott Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers and Chemistry*, 17:149–163, 1993.

Appendices

A Lexical Conventions

This section describes some of the conventions used by the CPL parsers. A few words of general advice: Use white space liberally — it makes programs easier to read and it reduces parsing errors.

A.1 Variables Names

CPL is extremely liberal when it comes to accepting names of variables. The following are all perfectly acceptable variable names: `X`, `2*x`, `2x-y^3`, `@!$%^’‘`, etc. Note that there is no requirement that the name of a variable must start with a letter. Thus just about anything can appear as part of the name of a variable. The exceptions are white space, brackets, and string quotes.

A.2 Label Names

CPL is extremely liberal when it comes to accepting names of labels. The only real requirement is that it start with the `#` sign. The following are all perfectly acceptable variable names: `#X`, `#2*x`, `#2x-y^3`, `#@!$%^’‘`, etc. Thus just about anything can appear as part of the name of a variable. The exceptions are white space, brackets, and string quotes.

A.3 Numbers

CPL uses `~` as the minus sign.

A.4 Comments

The start of a comment is indicated by a bang `!`, followed by a space. A comment extends all the way until the end-of-line character.

A.5 Reserved Words

Here are the reserved words and symbols in CPL.

```
{+} +} {|+|} +|} [+] +]  { } {| |} [ ] <- <-- <--- == >= => = <= < > ( )
ext if then else case of or otherwise let where in error to using true false
from readfile writefile primitive | & ( ) @ : ; , . o () process modify
{l+} {++} {l++} {|l+|} {|++|} {|l++|} [l+] [++] [l++] <<- <<-- <<--- {l {|l [l
```

The last row of symbols are used internally to annotate expressions with control information. Basically the `l` tag is used to indicate laziness, the `++` and `<<` for parallelism.

A.6 Precedence and Associativity

All binary operators, including user-defined macros used in infixed form, associate to the left. User-defined macros are lowest in precedence. The precedence of the other operators are listed below in increasing strength, with those having the same strength grouped into the same line.

```
{+} +} {|+|} +|} [+] +]
{l+} {|l+|} [l+]
{++} {|++|} [++]
{l++} {|l++|} [l++]
.
o
@
```

A.7 Embedded ML Codes

You can embed ML codes into a CPL script also. The start of a piece of embedded ML codes is indicated by a semi-colon, followed by a newline, followed by `! beginsml`, followed by a space, followed by a newline. The end of a piece of embedded ML codes is indicated by a newline, followed by `! endsml`, followed by a space, followed by a newline, followed by a semi-colon.

B Available Data Drivers

In this appendix I list the external data servers that have been registered for use in CPL. I group them by the Kleisli modules that implement them. See *The Kleisli Query System Reference Manual* for more information.

B.1 Entrez

The server interface to the National Center for Biotechnology Information GenBank database in ASN.1 format is implemented by these routines in the Kleisli module ENTREZSVR. Underlying it is the asncpl program written in C by Kyle Hart.

- `entrez-add: (#name: string, #level: num) -> string`

  Make connections to NCBI GenBank. For example, `entrez-add(#name: "entrez", #level: 2)` causes two connections to be made to NCBI GenBank and names both of these connections `entrez`. Then requests can be sent to either connections using the `process` REQUEST USING SERVERCONNECTION command of CPLlike this:

  ```
  process (#db: "na", #select: "small-scale", #path: "Seq-entry", #args: "")
  using entrez
  ```

  Requests to NCBI GenBank must have the form of a record `(#db:` DATABASE, `#select:` SELECTION, `#path:` PATH, `#args:` ADDITIONAL ARGUMENTS`)`. The DATABASE can be either `"na"` for nucleic acid, `"aa"` for amino acid, or `"ml"` for MEDLINE. The SELECTION should be a Boolean combination of index-value pairs or text terms. As records returned by NCBI GenBank is very complicated (over 12 levels of nesting and over 150 different kinds of subobjects), the PATH is used to prune these records by specifying the desired subtrees. See *The Kleisli Query System Reference Manual*, as well as NCBI documentation for Entrez, for more information.

- `entrez-del: string -> string`

  Close and remove the named connections. You can also use `delete-server` to do this; they have the same effect.

B.2 Sybase

The server interface to Sybase relational databases is implemented by these routines in the Kleisli module SYBASESVR. Underlying it is the sybcpl program written in C by Jonathan Crabtree.

- `sybase-add: (#name: string, #user: string, #password: string, #server: string, #level: num) -> string`

  Make connections to Sybase servers. You have to supply the proper user name, password, and server name. For example, `sybase-add(#name: "gdb", #user: "cbil", #password: "bogus",`

`#server: "WELCHSQL", #level: 2)` makes two connections to the Sybase system at John Hopkins University Welch Medical Library to access GDB. It names both connections `gdb` so that requests to them can be sent using the `process` REQUEST `using` SERVERCONNECTION command like this:

```
process "select * from locus l where l.locus_id = 12345"
using gdb
```

Requests to a Sybase server connection must be a string representing an SQL query. If the `#level` number is positive, then you can only send SQL queries satisfying a certain form down these connections. If the `#level` number is negative, then you can send any Sybase Transact-SQL command. The sign of the `#level` number is used to indicate to the CPL optimizer whether it is allowed to optimize the query — positive means optimization should be performed and the query is expected to be in a special form. The magnitude of the `#level` number indicates the number of connections or concurrency level.

A partial grammar for the restricted optimizable form of SQL is given below. Capitalized terms are rule names. Literals are enclosed in quotes.

```
TOP
  : UNIONQUERY

UNIONQUERY
  : QUERY
  | "(" UNIONQUERY ")"
  | UNIONQUERY "union" UNIONQUERY

QUERY
  : "select" FIELDS "from" TABLES "where" CONDS
  | "select distinct" FIELDS "from" TABLES "where" CONDS
  | "select * from" TABLE "where" CONDS


FIELDS
  : FIELD
  | FIELD "," FIELDS

FIELD
  : COLUMN_NAME "=" REF

TABLES
  : TABLE
  | TABLE "," TABLES

TABLE
  : TABLE_NAME RANGE_VARIABLE
```

```
  | DATABASE_NAME ".." TABLE_NAME RANGE_VARIABLE

CONDS
  : COND
  | "(" COND ")"
  | "not" CONDS
  | CONDS "and" CONDS
  | CONDS "or" CONDS

COND
  : REF "=" REF
  | REF "<=" REF
  | REF "<" REF
  | REF ">=" REF
  | REF ">" REF
  | REF "like" REF
  | REF "in" "(" QUERY ")"
  | "exists" "(" QUERY ")"

REF
  : RANGE_VARIABLE "." COLUMN_NAME
  | REF OP REF        /* OP is a recognized SQL operators such as + and -. */
  | BASE              /* BASE is any recognized SQL atoms such as integers. */
  | OP "(" REFS ")"   /* OP is a recognized SQL function such as substring. */

REFS
  : REF
  | REF "," REFS
```

- `sybase-views: (#name: string, #user: string, #password: string, #server: string, #level: num, #db: string, #prefix: string) -> string`

  Same as `sybase-add`, but you are also allowed to specify which database to import (the `db` field) and a prefix to add to the imported table names (the `prefix` field).

  For example, after doing

  ```
  sybase-views (#name: "gdb56",
    #user: "gdbsqllim", #password: "foobar", #server: "JICST",
    #level: 1, #db: "gdb", #prefix: "r-");
  ```

  you will be able to straight away say

  ```
  { x.#gdb_server_location | \x <- r-gdb_servers};
  ```

  to list the server locations given in the GDB table “gdb_servers.”

- `sybase-del: string -> string`

  Close and remove the named connection. You can also use `delete-server` to do this; they have the same effect.

- `sybase-connect: (#name: string, #user: string, #password: string, #server: string, #level: num) -> (#1: string -> { ''2 }, #2: '1 -> unit)`

  Make the indicated number of connections to the specified Sybase source. Return a pair of functions (f, g). Requests can be sent to these connections by applying f and connections can be shutdown by applying g.

- `sybase-xact: (#name: string, #user: string, #password: string, #server: string, #level: num) -> ((string -> { ''1 }) -> '2) -> '2`

  Similar to `sybase-connect`. `sybase-xact` X T works like this. It calls `sybase-connect` X to get the pair of functions (f, g). Then it executes the transaction $T(f)$, the result of the transaction is returned. Then it executes $g()$ to shutdown the connection.

B.3 Standard IO

The server interface for standard file input-output is implemented by these routines in the Kleisli module STDIO.

- `stdin: string -> '2`

  All server connections can be used as a function. `stdin` is the connection to the standard file input server. Thus you can think of `stdin ("afile")` as a shorthand for `process "afile" using stdin`. That is, read the file named `"afile"` and return its contents. Note that `stdin` assumes the actual file to be read as `"afile.val"` and that its type is described by the schema file `"afile.typ`.

- `stdout: (string, string, ''3) -> string`

  You can think of `stdout("afile", "{num}", {1,2,3,4})` as a shorthand for `process ("afile", "{num}", {1,2,3,4}) using stdout`. That is, create the file named `"afile"` of type `{num}` and write the set `{1,2,3,4}` to it. Note that `stdout` actually produces two files, `"afile.val"` and `"afile.typ"`. You should use `stdin` to read the contents of this file.

- `machineout: (string, string, ''3) -> string`

  Same as `stdout`, but write the file in Compressed CPL Standard Exchange Format. You should use `stdin` to read the contents of this file.

- `appendtofile: (string, string) -> string`

  Append a string to a file. Shorthand for `process` (FILENAME, MESSAGE) `using appendtofile`.

- `file-stringify: string -> string`

  Return the contents of the specified file as a string. Shorthand for `process` FILENAME `using file-stringify`.

- `file-intolines: string -> [ string ]`

  Return the contents of the specified file as a list of lines. Shorthand for `process` FILENAME `using file-intolines`.

B.4 Formatted Output

These routines allow output from CPL to be formatted in different ways. They are implemented in the Kleisli module FORMATTEDOUT.

- `htmlout: (string, string, ''3) -> string`

  Same as `stdout`, but ignore its second argument and use HTML format instead of the CPL Standard Exchange Format. It is actually a shorthand for `process` (FILENAME, FORMAT, DATA) `using htmlout`.

- `dynamicform: (string, string, ''3) -> string`

  Same as `htmlout`, but the second argument now indicates the output format to be used. Currently recognized output formats are `"html"`, `"standardcpl"`, and `"sgml"`. It is actually a shorthand for `process` (FILENAME, FORMAT, DATA) `using dynamicform`.

B.5 System Call

Making system call from CPL to Unix is handled by a server interface too. The routines of this interface are implemented in the Kleisli module SYSCALL.

- `syscall-co: (string, [ string ], string) -> ''3`

  The first argument to this function is the name of the Unix command to call. The second argument is the list of values to be used as commandline arguments. The third argument is a string to be used as the standard input stream to the Unix command. The Unix command is required to return a complex object in the CPL Standard Exchange Format in its standard output stream. This complex object is returned as the result of this function. This function is actually a shorthand for `process` (COMMAND, ARGUMENTS, INPUT) `using syscall-co`.

- `syscall-line: (string, [ string ], string) -> [ string ]`

  Same as `syscall-co`, but return the result of the system call as a list of lines. Also it does not require the Unix command to produce its result as a complex object. This function is actually a shorthand for `process` (COMMAND, ARGUMENTS, INPUT) `using syscall-line`.

- `syscall-string: (string, [ string ], string) -> string`

  Same as `syscall-line`, but return the result of the system call as a single string. This function is actually a shorthand for `process` (COMMAND, ARGUMENTS, INPUT) `using syscall-string`.

B.6 HTTP Call

Calling HTTP or World Wide Web servers from CPL could not be easier. The interface is implemented in the Kleisli module HTTPCALL.

- `httpcall-string: (string, string, string, string, string) -> string`

  The first argument is the name or IP address of the HTTP server you are calling. The second argument is the port number that the server is listening to. The third argument specifies the HTTP method to use; it must be either `"GET"` or `"POST"`. The fourth argument is the path of the WWW page you want to access. The last argument is a parameter string you want to pass to the HTTP server. It returns the WWW page as a string.

  For example, `httpcall-string ("corona.iss.nus.sg", "8080", "GET", "", "")` returns our BioInformatics Centre's homepage. Similarly, `httpcall-string ("corona.iss.nus.sg", "8080", "GET", "/examples/monitor/pallen/index.html", "")` brings back our BioInformatics Centre's page on our Protein Tyrosine Phosphatase Warehouse.

  It is also possible to run cgi-bin programs. To illustrate this, let me use the form "http://corona. iss. nus. sg:8080/ examples/ wwwlinks/ index.html." This form has one blank named "accn" that needs to be filled in. The filled form is then despatched to "http://corona. iss. nus. sg:8080/ cgi-bin/ examples/ wwwlinks/ run.pl" for execution using the HTTP POST method. Such a cgi-bin program that is designed to work with the HTTP POST method can be run by making a HTTP call like this: `httpcall-string ("corona.iss.nus.sg", "8080", "POST", "/cgi-bin/examples/wwwlinks/run.pl", "accn=M12345")`. It corresponds to running "http: //corona. iss. nus. sg:8080/ cgi-bin/ examples/ wwwlinks/ run.pl" with the "accn" argument set to "M12345."

  By the way, a cgi-bin program that is designed to work with the HTTP GET method can be run by making a HTTP call like this: `httpcall-string ("corona.iss.nus.sg", "8080", "GET", "/cgi-bin/examples/wwwlinks/run.tcl", "M12345")`. It corresponds to "http://corona. iss. nus. sg:8080/ cgi-bin/ examples/ wwwlinks/ run.tcl?M12345."

- `httpcall-line: (string, string, string, string, string) -> [string]`

  This one is same as `httpcall-string`, except that it returns its output as a list of strings.

B.7 Hwee Hwa's Indexing and Matching Facilities

My colleague, Pang Hwee Hwa, at the Institute of Systems Science developed SeqIndex, a suite of programs for indexing and matching long strings like amino acid and DNA sequences. The SeqIndex suite currently consists of the following five programs: loadseq—for creating a sequence index; scanseq—for finding sequences having patterns specified by a regular expression, modulo edit distance; signature—for detecting conserved patterns in sequences; duplicate—for detecting repeated substrings modulo edit distance in sequences; and dnaduplicate—for detecting tandem and inverted repeats modulo edit distance in DNA sequences. The interface to the SeqIndex suite is implemented by routines in the Kleisli module SEQINDEXSVR.

- `seqindex-out: (string,string,{(#uid: num,#seq:string; '1)}) -> string`

  Make a connection to the loadseq program in the SeqIndex suite. This function is for creating a sequence index. The function call `seqindex-out`(F, T, S) is equivalent to the server call `process` S `to` F `using seqindex-out`. In fact, it is normally invoked like this:

  ```
  writefile
   { (#uid: 1, #seq: "PROTEIN-1"),
     (#uid: 2, #seq: "PROTEIN-2"),
      ...,
     (#uid: 100, #seq: "PROTEIN-100") }
  to "myindexfile" using seqindex-out;
  ```

  which builds an index `"myindexfile"` for my 100 protein sequences. Such an index file can then be used by `seqindex-scanseq`, and `seqindex-signature`. Note that the numbers in the `#uid` field must be unique.

- `seqindex-setdb: (string,string,{(#uid: num,#seq:string; '1)}) -> string`

  Same as `seqindex-out`.

- `seqindex-pressdb: (string,string,{(#uid: num,#seq:string; '1)}) -> string`

  Same as `seqindex-out`, but works on DNA sequences only.

- `seqindex-in: string -> {(#uid: num, #seq: string) }`

  You can think of `seqindex-in "myindexfile"` as a shorthand for `process "myindexfile" using seqindex-in`. That is, read the contents of the sequence index `"myindexfile"`.

- `seqindex-scanseq: (#name: string, #index: string, #level: num) -> string`

  Makes connections to the scanseq program in the SeqIndex suite. You have to supply the name of the index to use. For example, `seqindex-scanseq (#name: "ndx", #index: "myindexfile", #level: 2)` makes two connections to scanseq and names both connections `ndx` so that requests to them can be sent using the `process` REQUEST `using` SERVERCONNECTION command. There are three kinds of request you can send to scanseq:

  - `process <#list: ()> using ndx;`

    This kind of request lists the contents of the index. Thus the above has the same effect as `process "myindexfile" using seqindex-in`. The output of this request has type `{(#uid: num, #seq: string)}`.

  - `process <#get: 3> using ndx;`

    This kind of request retrieves the set of sequences in the index whose associated `#uid` field has the specified value. Using our running example, the above returns the set `{(#uid: 3, #seq: "PROTEIN-3")}`.

  - `process <#doit: (#restrict: true, #pattern: "[LIV].GLGF{3,5}", #edit: 2, #overlap: 30, #favourshort: true)> using ndx;`

    This kind of request retrieves the positions in which a given pattern appears in sequences in the index. In our example, the pattern is `"[LIV].GLGF{3,5}"`. This pattern has the same semantics as in Perl—`[LIV]` means a letter that can be any of L, I, or V; `.` means any single character; and `F{3,5}` means the letter F repeated 3 to 5 times. Thus, `LRGLGFFFF` satisfies the pattern, while `RRGLGFFFF` does not.

The other fields in the request control the strictness of pattern matching. The `#edit` field specifies the number of mismatched characters allowed. The `#restrict` field specifies the places where mismatching are allowed: if it is set to true, then mismatch is only allowed in "ambiguous" places like `[LIV]` but not in "definite" places like `GLGF`; if it is set to false, then mismatch is allowed anywhere. Sometimes two matched substrings physically overlap. It is desirable to report only one of them. The selection of which one to report is controlled by the `#overlap` and `#favourshort` fields. In our example, if two matched substring overlap at least 30 characters then only the shorter one is reported. Tie is broken by favouring the one that occurs more to the left.

The output of this kind of request has type `{(#uid: num, #start-pos: num, #end-pos: num, #start: [ num ], #dist: num)}`. Each entry in the output reports the position of a match. The `#uid` is identifies the sequence that is matched. The `#start-pos` indicates the starting position of the match. The `#end-pos` indicates the ending position of the match. The `#dist` is the edit distance of the match. The `#start` is a list of numbers indicating the start positions of various components of the pattern—the detail of these components need not concern us here.

- `seqindex-stdprositescan: (#name: string, #level: num) -> string`

  Makes connections to the prositescan program in the SeqIndex suite and sets it up to use the default Prosite motif database. For example, `seqindex-stdprositescan: (#name: "stdprosite", #level: 1)` makes one connection to prositescan and names it `stdprosite`. Requests can then be sent using the `process` REQUEST `using` SERVERCONNECTION command. A request to prositescan always has the form:

  - `process (#restrict: true, #edit: 1, #fussy: true, #seq: "PROTEIN-1") using stdprosite;`

    Here `"PROTEIN-1"` is your protein sequence. Every motif in the default Prosite motif database will be checked against your protein sequence and the hits will be reported. There are two kinds of motifs: rare motifs and common motifs. If the `#fussy` field is true, then only rare motifs are considered; otherwise all motifs are considered.

    In contrast to the standard Swiss PrositeScan program, Hwee Hwa's prositescan also supports matching modulo edit distance. The `#edit` field specifies the allowed edit distance and the `#restrict` field specifies positions where mismatches are allowed.

    The type of the output looks like this: `{(#pattern: string, #name: string, #dist: num, #prositeid: string, #docid: string, #description: string, #start-pos: num, #end-pos: num, #start: [ num ]) }`. Each record represents a motif hit. The `#pattern` field is the motif from the motif database. The `#name` field is the name of the motif. The `#prositeid` field is the Prosite identifier of the motif. The `#docid` field is the Prosite document identifier of the motif. The `#description` field is a short description of the motif. The `#dist` field is the edit distance of the match to the motif. The `#start-pos` field is the start position of the match. The `#end-pos` field is the end position of the match. The `#start` field is a list of numbers indicating the positions of various components of the motif in the match.

- `seqindex-prositescan: (#name: string, #prosite: string, #level: num) -> string`

  Same as `seqindex-prositescan`. However, you can specify your own motif database in the `#prosite` field.

- `seqindex-signature: (#name: string, #index: string, #optfortime: bool, #level: num) -> string`

  Makes connections to the signature program in the SeqIndex suite. you have to supply the name of the index to use. For example, `seqindex-signature (#name: "sign", #index: "myindexfile", #optfortime: true, #level: 1)` makes a connection to signature and calls that connection `sign`. Set the `#optfortime` field to true to instruct the signature program to trade space for time. Requests can then be sent using the `process` REQUEST `using` SERVERCONNECTION command. There are three kinds of request:

  - `process <#list: ()> using sign;`

    Lists the contents of the index. Thus the above has the same effect as `process "myindexfile" using seqindex-in`. The output of this request has type `{(#uid: num, #seq: string)}`.

  - `process <#get: 2> using sign;`

    This kind of request retrieves the set of sequences in the index whose associated `#uid` is the specified value. Using our running example, the output of this request is the set `{(#uid: 2, #seq: "PROTEIN-2")}`.

  - `process <#doit: (#min: 10, #max: 20, #edit: 3, #support: 60)> using sign;`

    This kind of request returns patterns of specified length that are conserved by sequences in the index. The `#min` and `#max` fields specifies the length of the patterns to consider. The `#edit` field specifies the allowable edit distance between different occurrences of the same pattern. The `#support` field specifies the minimum percentage of the sequences in the index that a pattern must cover before the pattern can be considered conserved. Note that the value specifed for `#edit` should be strictly less than `#min`.

    The output of this kind of request has the type `{ (#pattern: string, #count: n, #percent: num) }`. Each record reports a conserved pattern. The `#pattern` is the conserved pattern. The `#count` is the number of sequences this pattern occurs in. The `#percent` is the percentage of sequences this pattern occurs in.

- `seqindex-duplicate: (#name: string, #noduplicate: bool, #optfortime: bool, #level: num) -> string`

  Makes connections to the duplicate program in the SeqIndex suite. Set the `#noduplicate` field to true to instruct duplicate to ignore repeats that are subsumed by longer repeats. Set the `#optfortime` field to true to instruct duplicate to trade space for time. Set the `#level` field to indicate the number of connections to make. For example, `seqindex-duplicate (#name: "dup", #noduplicate: false, #optfortime: true, #level: 2)` makes two connections to duplicate and names both of them `dup`. We can send request to signature using the `process` REQUEST `using` SERVERCONNECTION command. A request must have the form:

  - `process (#min: 5, #max: 11, #incr: 2, #support: 2, #edit: 0, #space: 10, #seq: "PROTEIN-3") using dup;`

Here `"PROTEIN-3"` is your protein sequence. This request returns all substring of lengths 5, 7, 9, 11—-`#min` is 5, `#max` is 11, `#incr` is 2—that appear at least 2 times in your protein—`#support` is 2—at least 10 residues apart—`#space` is 10—without any mutation—`#edit` is 0.

The output has the type `{(#pattern: string, #count: num, #start: [ num ])}`. Each record represents a repeat. The `#pattern` field is the repeated pattern. The `#count` field is the number of times it repeated. The `#start` field is the a list of numbers indicating the start positions of each occurence of the repeat.

- `seqindex-dnaduplicate: (#name: string, #noduplicate: bool, #optfortime: bool, #level: num) -> string`

  Connects to the dnaduplicate program in the SeqIndex suite. General effect is same as `seqindex-dnaduplicate` but works on DNA instead of protein sequences and detects both tandem and inverted repeats.

  The requests that you sent to a dnaduplicate connection have the same form as that to a duplicate connection.

  The output, however, is slightly different and has this type: `{(#pattern: string, #count: num, #start: [ num ], #inv_start: [ num ])}`. The `#start` field is a list of numbers indicating the start of tandem repeats. The `#inv_start` field is a list of numbers indicating the start of inverted repeats.

B.8 Steve Brenner's Taxonomy Driver

Steve Brenner passed me a driver for NCBI's taxonomy database when he visited me in July 1997. The taxonomy is organized as a tree (what else?!) Each node of is identified by an id such as 3750 that is unique to the tree. Associated with each node are: a set of names—these are the scientific and common names such as "apples" and are not necessarily unique; a rank such as "species"; a parent node id such as 3749, and a set of children. (Question: who is the parent of the root of the taxonmy tree? Answer: that's the root itself!) The interface to Steve's driver is in the module TAXONSVR. Here are the functions provided by the interface.

- `phy-add: string-> string`

  Establish a connection to the taxonomy driver. For example, `phy-add "phy"` makes a connection to the taxonomy driver and calls it `phy`. Requests to the taxonomy driver can then be made using the `process` REQUEST `using` SERVERCONNECTION command. There are four kinds of requests:

  - `process <#nameid: "apple"> using phy;`

    Return the set of all id's whose names are `"apple"`. The output thus has type `{num}`.

  - `process <#idinfo: 3749> using phy;`

    Return a set containing information on the node whose id is 3749. The output corresponding to the example above looks like:

```
{(#name: {"Malus"},
  #id: 3749,
  #rank: "genus",
  #parent: 3745,
  #child: {28521, 3750, 3751, 3752})}
```

  That is, the type of the output is `{(#name: {string}`, `#id: num`, `#rank: string`, `#parent:num`, `#child: {num})}`.

  - `process <#lsid: ()> using phy;`

    Return a set containing all the id's in the taxonomy database. The type of the output is `{num}`.

  - `process <#lsname: ()> using phy;`

    Return a set containing all the names in the taxonomy database. The type of the output is `{ string}`.

- `phy-del: string -> string`

  Delete the named connection to the taxonomy driver. Same effect as `delete-server`.

B.9 Steve Brenner's SCOP Driver

Steve Brenner passed me a driver to his SCOP database [24] when he visited me in July 1997. The SCOP database is classification of protein families derived from sequence as well as structural similarity. Each entry is SCOP is identified by a sid string such as "d1sert_" and corresponds to a domain in a PDB entry. These entries are organized into a classification hierarchy. Each node in the hierarchy is identified by a node number such as "1.010.002.001.002.003". Several sid's can live in the same node, because there are classified to belong to the same family. Associated with each sid are the name of the PDB source of that entry such as "1ser"; the region that entry corresponds to in the PDB source such as "t:"; the corresponding node number of the SCOP hierarchy; the type and description of the entire path from the root to that node of the SCOP hierarchy.

The SCOPSVR module interfaces to Steve's driver. Here are the functions provided by it.

- `scop-add: string-> string`

  Establish a connection to the SCOP driver. For example, `scop-add "scop"` makes a connection to the SCOP driver and calls it `scop`. Requests to the SCOP driver can then be made usiong the `process` REQUEST `using` SERVERCONNECTION command. there are five types of request:

  - `process <#sidinfo: "d1sert_"> using scop;`

    Return a set containing information associated with the give sid. The output for the example above looks like this:

```
{(#sid: "d1sert_",
  #pdb: "1ser",
  #region: "t:",
```

```
        #num: "1.010.002.001.002.003",
        #type: (#cl: "1.010",
                #cf: "1.010.002",
                #sf: "1.010.002.001",
                #fa: "1.010.002.001.002",
                #sp: "1.010.002.001.002.003"),
        #desc: (#cl: "Class: Non-protein",
                #cf: "Fold: Nucleic acids",
                #sf: "Superfamily: Complexed with protein",
                #fa: "Family: tRNA",
                #dm: "Protein: tRNA",
                #sp: "Species: <i>synthetic</i>"))}
```

  - `process <#sidallinfo: ()> using scop;`
    Return information on all sid's.
  - `process <#lssid: ()> using scop;`
    Return a set of containing all the sid's. The type of the output is `{string}`.
  - `process <#numsid: "1.010.002.001"> using scop;`
    Return a set of strings containing all sid's living in the specified node of the SCOP hierarchy. These are sid's in the same class, fold, superfamily, family, protein, or species, depending on the specified node number.
  - `process <#pdbsid: "1ser"> using scop;`
    Return a set of stringd containing all sid's associated with the specified PDB entry. These are all the domains in that PDB entries.

- `scop-del: string -> string`

  Close and remove the named connection. You can also use `delete-server` to do this. They have the same effect.

B.10 Warren Gish's WU-BLAST2

Warren Gish's Washington University BLAST or WU-BLAST2 is a software suite for rapid and sensitive similarity searches of protein and nucleotide sequence databases [1, 20]. This suite includes the following programs: setdb—create a protein sequence database for BLASTing, pressdb—create a DNA sequence database for BLASTing, blastp—BLAST a protein sequence against a protein sequence database, blastn—BLAST a DNA sequence against a DNA sequence database, blastx—BLAST a DNA sequence translated in all reading frames against a protein sequence database, tblastn—BLAST a protein sequence against a DNA sequence database dynamically translated in all reading frames, tblastx—BLAST the six-frame translations of a DNA sequence against the six-frame translations of a DNA sequence database.

The interface to the driver for WU-BLAST2 is in the module LocalBLAST. Here are the functions provided by the interface.

- `localblast-pressdb : (#1: string, #2: string, #3: {(#uid: num, #accession: string, #title: string, #seq: string; '3) }) -> string`

  Make a connection to the pressdb program in the WU-BLAST2 suite. This function is for creating a DNA sequence database for BLASTing. The function call `localblast-pressdb`(F, T, S) is equivalent to the server call `process` S `to` F `using localblast-pressdb`. The database F thus prepared can then be using for subsequent BLASTing.

- `localblast-setdb : (#1: string, #2: string, #3: {(#uid: num, #accession: string, #title: string, #seq: string; '3) }) -> string`

  Make a connection to the setdb program in the WU-BLAST2 suite. This function is for creating a protein sequence database for BLASTing. The function call `localblast-setdb`(F, T, S) is equivalent to the server call `process` S `to` F `using localblast-setdb`. The database F thus prepared can then be using for subsequent BLASTing.

- `localblast-tblastx : (#name: string, #db: string) -> string`

  Make a connection to the tblastx program in the WU-BLAST2 suite. You have to supply the database that you want to BLAST against. This connection produces summary of BLAST hits only, which is useful if you just want to know the score of the hits.

  For example, `localblast-tblastx (#name: "myblast", #db: "mydb")` makes a server connection `myblast` available. After you have done the above, you can send your sequence to be TBLASTXed against `mydb` like this:

  ```
  process "YOUR QUERY SEQUENCE" using myblast;
  ```

  The output of this has type `{(#uid: num, #accession: string, #title: string, #score: num, #pscore: num, #identities: num, #positives: num) }`. Each record in the output is a summary of a hit. The `#score` is the raw score of the hit. The `#pscore` is the p-score of the hit. The `#identities` is the percentage of identities in the hit. The `#positives` is the percentage of positives in the hit.

- `localblast-tblastn : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx`, but connect to tblastn instead.

- `localblast-blastx : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx`, but connect to blastx instead.

- `localblast-blastn : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx`, but connect to blastn instead.

- `localblast-blastp : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx`, but connect to blastp instead.

- `localblast-tblastx-detail : (#name: string, #db: string) -> string`

  Make a connection to the tblastx program in the WU-BLAST2 suite. You have to supply the database that you want to BLAST against. This connection produces complete report of hits.

For example, `localblast-tblastx-detail (#name: "myblast", #db: "mydb")` makes a server connection `myblast` available. After you have done the above, you can send your sequence to be TBLASTXed against `mydb` like this:

```
process "YOUR QUERY SEQUENCE" using myblast;
```

The output of this has type `{(#uid: num, #accession: string, #title: string, #hits: [ (#score: num, #pscore: num, #identities: num, #positives: num, #querystart: num, #queryend: num, #subjectstart: num, #subjectend: num, #alignment: string)])}`. Each record in the output is a summary of a sequence in `mydb` that was hit by `"YOUR QUERY SEQUENCE"`. The `#score` is the raw score of the hit. The `#pscore` is the p-score of the hit. The `#identities` is the percentage of identities in the hit. The `#positives` is the percentage of positives in the hit. The `#querystart` and `#queryend` mark the hit position in `"YOUR QUERY SEQUENCE"`. The `subjectstart` and `#subjectend` mark the hit position in the sequence in the database that was hit. The `#alignment` is a three-line string representing the alignment of this hit.

- `localblast-tblastn-detail : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx-detail`, but connect to tblastn instead.

- `localblast-blastx-detail : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx-detail`, but connect to blastx instead.

- `localblast-blastn-detail : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx-detail`, but connect to blastn instead.

- `localblast-blastp-detail : (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx-detail`, but connect to blastp instead.

B.11 NCBI's BLAST

The National Center for Biotechnology Information in Washington DC provides a web-based service for rapid and sensitive similarity searches of protein and necleotide sequences stored there. Their BLAST programs (blastp, blastn, blastx, tblastn, tblastx) are different from Gish's. The interface to this web-based service is in the module WEBBLAST. Here are the functions provided by the interface.

- `webblast-blastp: (#name: string, #db: string, #level: num) -> string`

  Make connections to the blastp program in the National Center for Biotechnology Information website. The number of connections is given by the `#level` field. You must also specify the database you want to BLAST against. This database is given by the `db` field. These connections produce summary of BLAST hits only.

  For example, `webblast-blastp (#name: "myblast", #db: "nr", #level: 1)` makes a server connection `myblast` available. After you have done this, you can send your sequence to be BLASTPed against the `nr` database at the National Center for Biotechnology Information like this:

```
process "YOUR QUERY SEQUENCE" using myblast;
```

The output of this has type `{(#uid: num, #accession: string, #title: string, #score: num, #p-n: num, #n: num)}`. Each record in the output is a summary of a hit. The `#p-n` is the p-score of the hit.

- `webblast-blastn: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp`, but connect to blastn instead.

- `webblast-blastx: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp`, but connect to blastx instead.

- `webblast-tblastx: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp`, but connect to tblastx instead.

- `webblast-tblastn: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp`, but connect to tblastn instead.

- `webblast-blastp-detail: (#name: string, #db: string, #level: num) -> string`

  Make connections to the blastp program in the National Center for Biotechnology Information website. The number of connections is given by the `#level` field. You must also specify the database you want to BLAST against. This database is given by the `db` field. These connections produce complete report of BLAST hits.

  For example, `webblast-blastp-detail (#name: "myblast", #db: "nr", #level: 1)` makes a server connection `myblast` available. After you have done this, you can send your sequence to be BLASTPed against the `nr` database at the National Center for Biotechnology Information like this:

  ```
  process "YOUR QUERY SEQUENCE" using myblast;
  ```

  The output of this has type `{(#uid: num, #accession: string, #title: string, #hits: [(#score: num, #pscore: num, #identities: num, #positives: num, #querystart: num, #queryend: num, #subjectstart: num, #subjectend: num, #alignment: string)])}`. The interpretation is similar to that of the WU-BLAST2 interface.

- `webblast-blastn-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp-detail`, but connect to blastn instead.

- `webblast-blastx-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp-detail`, but connect to blastx instead.

- `webblast-tblastx-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp-detail`, but connect to tblastx instead.

- `webblast-tblastn-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast-blastp-detail`, but connect to tblastn instead.

B.12 Wootton and Federhen's SEG Program

Wootton and Federhen [42] implemented the SEG program for identifying and masking segments of low compositional complexity in amino acid sequences. The interface to this program is in the module SEGFILTER. Here is the function provided by this module.

- `segfilter :string-> [(#highcomplexity: bool, #start: num, #end: num, #seq: string)]`

  The input to this function is an amino acid sequence. The output is a list of regions alternating between high and low complexity. Each region is a record having four fields. The `#highcomplexity` field is true iff the region has high complexity according to Wootton and Federhen [42]. The `#start` field is the start position of the region. The `#end` field is the end position of the region. The `#seq` field is the actual amino acid sequence within the region.

B.13 Chen Jing's cluster Program

Chen Jing implemented a suite of clustering programs after some discussion with Pang Hwee Hwa, Daphna Strauss, and myself. His first program, cluster, reads lines of numbers. Numbers on the same lines are considered to be related. Numbers on two lines that share some common numbers are also considered to be related. The program simply groups related numbers into the same sets and print these sets out.

His second program, cluster2, takes a threshold number T from the commandline, then reads lines of numbers. iIt also does grouping. Two groups A and B are merged only if $(|A \cap B|)/\max(|A|, |B|) > T \times (1 - 1/\max(|A|, |B|))$. If two groups A and B have a small intersection, but below the threshold, the intersection is deleted from the bigger group.

His third program, cluster5 (yes, cluster5!), is similar to cluster2. However, if two grooups A, B have small intersection but below threshold, the intersection is deleted from the smaller group.

His fourth program, expand, takes a expansion limit, then reads lines of numbers. Each line of numbers is interpreted as specifying edges in a graph: $\{(x, y) \mid l$ is a line of numbers, x is the first number of l, y is any other numbers of $l\}$. An expansion step calculates what new numbers can be reached by following one more edge: $\{(x, z) \mid l$ is a line of numbers, x is first number of l, y is some other number of l, l' is also a line of numbers, y is also the first number of l', z is some other number of $l'\}$. His program keeps doing expansion until it reaches the given limit. However, if the limit is 0, it goes on expanding until nothing new can be reached.

His fifth program, cluster3 (yes, cluster3!), takes a threshold, an expansion limit, and a clustering limit, then reads lines of numbers. These lines of numbers are interpreted in the same way as the expand program. The threshold is interpreted in the same way as the cluster2 program. This cluster3 program simply alternates between expand and cluster2 until the clustering limit is reached or until it reaches a stable state.

The interface to these programs is provided by the CLUSTER module. Here are the functions provided by this module.

- `cluster : { { num } } -> { (#rep: num, #cluster: { num }) }`

  Given a set of set of numbers $\{S_1, \ldots, S_n\}$. All numbers in the same set S_i are deemed related. If $S_i \cap S_j \neq \{\}$, then numbers in S_i and S_j are deemed related. Related numbers are then grouped into clusters C_1, ..., C_m, such that any two numbers from any two clusters are not related and all numbers in the same cluster are related. Representatives r_1, ..., r_m are chosen for these clusters. Here is how a representative r for a cluster C is chosen. First r must be a member of the C. Second r must appear in at least as many S_i as any other member of C. These clusters and representatives are then returned.

  The `cluster` function is actually a server request. So you can also invoke like:

  ```
  process {{ 1, 2, 3}, {3,4,5}, {6,7,8}, {7,9,0}} using cluster;
  ```

  The output is:

  ```
  {(#rep: 3, #cluster: {1, 2, 3, 4, 5}),
   (#rep: 7, #cluster: {0, 6, 7, 8, 9})}
  ```

- `cluster2-add: (#name: string, #threshold: num) -> string`

  Set up a connection to Chen Jing's cluster2 program. Set its threshold to the specified number. Name this connection using the specified name. For example,

  ```
  cluster2-add (#name: "c2", #threshold: "0.8");
  ```

  causes a server connection `c2: {{num}}->{(#rep: num, #cluster: {num})}` to be set up. You can then make cluster2 request by calling `c2` the same way you call `cluster`.

- `cluster5-add: (#name: string, #threshold: num) -> string`

  Ditto.

- `cluster3-add: (#name: string, #threshold: num, #expandtimes: num, #cluster2times: num) -> string`

  Ditto.

- `expand-add: (#name: string, #expandtimes: num) -> string`

  Ditto.

B.14 Virtual Genome Centre's ORF Translation

The Virtual Genome Centre has a program for identifying open reading frames in DNA sequences. The method is based on [31]. The program is at `http://alces.med.umn.edu/webtrans.html`. Dapha Strauss developed a driver called `weborf` to this program. The interface to Daphna's program is in the module WEBORF.

- `weborf-add: (#name: string, #min: num, #level: num, #args: { <#standard: unit, #candida: unit, #any: unit, #met: unit, #all: unit, #unique: unit, #ignore: unit, #random: unit> }) -> string`

  Set up connections to Daphna's `weborf` program. The number of connections is given by the `#level` field. All these connections are given the name in the `#name` field. The `#min` field specifies the minimum length on ORFs. The `#args` field specifies the optional arguments to be passed to `weborf`. For example,

  ```
  weborf-add (#name: "findorf", #level: 1, #min: 100, #args: { });
  ```

  sets up a connection `findorf` to which you can make requests to `weborf` for finding ORFs of minimum length 100, using default parameters. The parameters will be described shortly. After the above, you can send your request by doing

  ```
  process YOUR-DNA-SEQUENCE using findorf;
  ```

  The output of your request has type `{(#left: num, #right: num, #frame: num, #aa-size: num, #seq: string)}`. Each record in the output is an ORF predicted in your dna sequence. The `#left` field is the starting point of the translation frame. The `#right` field is the ending point. The `#frame` is the translation frame. The `#aa-size` is the length of the translated protein. The `#seq` is the translated protein sequence.

  Okay, let's talk about the parameters you can specified in the `#args` field now. These are

  - `<#standard:()>`. Use standard genetic code. This is default.
  - `<#candida:()>`. Use Candida genetic code.
  - `<#any:()>`. Use any start codon.
  - `<#met:()>`. Use only Met as start codon. This is default.
  - `<#all:()>`. Return all predicted ORFs. (They may overlap.)
  - `<#unique:()>`. Return only non-overlapping ORFs. This is default.
  - `<#ignore:()>`. Skip ambiguous positions. Ie., skip all the "N"s.
  - `<#random:()>`. Randomly substitute ambiguous positions. This is default.

B.15 Gapped BLAST and PSI-BLAST

Altschul et. al. [2] announced their next generation of BLAST programs called Gapped BLAST and PSI-BLAST. Wang Jiren wrote the Kleisli/CPL driver to them The interface to this driver is in the module GAPBLAST. It is very similar to the interface to WU-BLAST in the module LocalBLAST.

- `gapblast-setdb: (#1: string, #2: string, #3: { (#uid: num, #accession: string, #title: string, #seq: string; '3) }) -> string`

  Similar to `localblast-setdb`. But note that the protein BLAST indices prepared by these two commands are not interchangeable.

- `gapblast-pressdb: (#1: string, #2: string, #3: { (#uid: num, #accession: string, #title: string, #seq: string; '3) }) -> string`

  Similar to `localblast-pressdb`. But note that the dna BLAST indices prepared by these two commands are not interchangeable.

- `gapblast-blastp: (#name: string, #db: string) -> string`

  Similar to `localblast-blastp`.

- `gapblast-blastn: (#name: string, #db: string) -> string`

  Similar to `localblast-blastn`.

- `gapblast-blastx: (#name: string, #db: string) -> string`

  Similar to `localblast-blastx`.

- `gapblast-tblastx: (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx`.

- `gapblast-tblastn: (#name: string, #db: string) -> string`

  Similar to `localblast-tblastn`.

- `gapblast-psi: (#name: string, #db: string) -> string`

  Similar to `localblast-blastp`. According to [2], `gapblast-psi` is slower but is more sensitive than `gapblast-blastp`.

- `gapblast-blastp-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-blastp-detail`.

- `gapblast-blastn-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-blastn-detail`.

- `gapblast-blastx-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-blastx-detail`.

- `gapblast-tblastx-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-tblastx-detail`.

- `gapblast-tblastn-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-tblastn-detail`.

- `gapblast-psi-detail: (#name: string, #db: string) -> string`

  Similar to `localblast-blastp-detail`. According to [2], it `gapblast-psi` is slower but is more sensitive than `gapblast-blastp`.

B.16 ACeDB

This subsection was written by Oliver Wu Peng.

ACeDB [8] is an acronym for A Caenorhabditis elegans Database. It can refer to a database and data concerning the nematode C. elegans, or to the database software alone. This document is concerned primarily with the latter meaning. ACeDB is being adapted by many groups to organize molecular biology data about the genomes of diverse species. ACeDB was written and developed by Richard Durbin (MRC LMB Cambridge, England) and Jean Thierry-Mieg (CNRS, Montpellier, France), beginning in 1989. It is written in the C programming language and uses the X11 windowing system to provide a platform independent graphical user interface.

we implemented an ACeDB driver for Kleisli. It accepts normal ACeDB (version 3 and above) databases queries, processes queries and generates queries results in CPL formats. The interface to ACeDB driver is in the module ACESVR. Underlying is the program `ace2cpl` written in C. Here are the functions provided by this module:

- `acedb-add: (#name: string, #db: string, #level: num) -> string`

  Establish connections to the ACeDB driver. For example, `acedb-add (#name: "ace", #db: "/acedb/ace4_5", #level: 2)` makes two connections to a local ACeDB (version 4.5) database located at `"/acedb/ace4_5"` and calls it `ace`. Queries to the ACeDB driver can then be made by using the `process <#query: QUERY> using` SERVERCONNECTION command. For example,

  ```
  process <#query: "find pathway al*"> using ace
  ```

  will return a set containing information about those pathway names begin with `al`.

- `acedb-del: string -> string`

  Close and remove the named connections. Same effect could be made if `delete-server` is applied.

B.17 Erik Sonnhammer's hmmPfam

Erik Sonnhammer created the package hmmPfam [32] that uses Sean Eddy's HMMER package [9, 11, 10] to search Pfam; see `http://www.sanger.ac.uk/Pfam`.

Wang Jiren implemented a driver to hmmPfam. The interface to Jiren's driver is in the module HMMPFAM. The functions provided by this module are:

- `hmmpfam-summary: string -> { <#hmmls: (#hmmfile: string, #hits: { (#bitsscore: num, #start: num, #end: num) }), #hmmfs: (#hmmfile: string, #hits: { (#score: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num)})>}`

  Similar to `hmmpfam-detail` below, but its output does not contain any detail of alignment.

- `hmmpfam-detail: string -> { <#hmmls:(#hmmfile: string,#hits: { (#bitsscore: num, #start: num, #end: num, #alignment: string) }), #hmmfs: (#hmmfile: string, #hits: { (#score: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num, #alignment: string)})>}`

  Use this function to submit your protein sequence to hmmPfam for analysis. This is a server call, so you can also invoke it like this:

  ```
  process "MGKIIGIDLGTTNSC...QQAADDVVDAEFKEVKKD" using hmmpfam-detail;
  ```

  The result is returned like this, where the variant tag of each item in the result indicates which hmmer program is used:

  ```
  {<#hmmfs: (#hmmfile: "HSP70.HMM",
             #hits: {(#score: 1259.65,
                      #seqstart: 3,
                      #seqend: 606,
                      #hmmstart: 0,
                      #hmmend: 624,
                      #alignment: "VIGIDLGTTYS...
                                   +IGIDLGTT+S...
                                   IIGIDLGTTNS...")})>}
  ```

  By the way, you can get more information on a particular hit from the Pfam server in England. For example, the hmmfile `HSP70.HMM` can be retrieved from England using the URL `http://www.sanger.ac.uk/Pfam/desc/HSP70.shtml`.

B.18 Sean Eddy's hmmer

Hidden Markov models (HMMs) can be used to do multiple sequence alignment and database searching, using statistical descriptions of a sequence family's consensus. They can align very large numbers of sequences (thousands). Database search sensitivity is, in many cases, as sensitive as structure-based ”inverse folding” methods such as threading. The HMMER suite of programs by Sean Eddy is a nice implementation doing this [9, 11, 10].

Wang Jiren wrote a driver to HMMER. The interface for this driver is in module HMMER. It provides the functions listed below:

- `hmm-localsearch: (#name: string, #model: string, #threshold: num, #level: num, #dna: bool, #window: num, #best: bool) -> string`

  Establish connections to HMMER. The number of connections is specified in the `#level` field. The name to be used is specified in the `#name` field. The threshold (report only matches having bitscore/score above this number) is specified in the `#threshold` field; the suggested value is 0. Set the `#dna` field to `true` if you want to search (complementary strand of) DNA sequences, Set `#window` to the desired scanning window length (ie. the maximum matched subsequence size);

a good number is 1000. Set `#best` to `true` if you want only the single best local match to be reported. Set `#model` to the HMM file of the HMM you want to match against.

Here is an example on how to establish a connection to HMMER:

```
hmm-localsearch (
  #name: "hmmer",
  #model: "/usr49/BIC/hmm/hmmPfam/HSP70.HMM",
  #threshold: 0, #window: 1000, #level: 1,
  #dna: false, #best: false);
```

After doing the above, you can match against the HSP70 HMM by sending request to the `hmmer` connection in the following ways:

- `process <#hmms: "AFGIDLGTT...LGQTAGE" > using hmmer;`

  Try to match the sequence `"AFGIDLGTT...LGQTAGE"` to the given HMM (HSP70). A complete match of the HMM to the sequence is required; i.e., hmms does a global Needleman/Wunsch style alignment.

  This type of request is designed to be useful for detecting complete proteins (or, less commonly, complete nucleic acid sequences) in sequence databases. The requirement for a complete match to the model means that it is useful only for whole-sequence models; for instance, a globin HMM will recognize other com- plete globin sequences (though it will not recognize the few proteins with multiple globin domains!) The output has type `{ (#hmmfile: string, #hits: { (#bitsscore: num, #start: num, #end: num, #alignment: string)})}`.

  The scores reported are in bits of information. Specifically, they are log-odds scores: the log of the ratio of the probability of the sequence given the model and the probability of the sequence given a simple random sequence model. This score is related to the statistical significance of the alignment. A score of zero is marginal; according to the model's statistics, it's 50-50 likely that the alignment is a real match to the model, and 50-50 likely that it's not. The higher the score, the better; a score of 100 means that it is 2^100-fold more likely that the sequence is a match to the model than not. In practice, a database contains many more unrelated sequences than related ones, so the actual score required for statistical significance is somewhat higher than zero.

- `process <#hmmls: "AFGIDLGTT...LGQTAGE" > using hmmer;`

  This type of request is designed to be particularly useful for detecting complete repeated motifs in long sequences. It has been used successfully for Alu detection in human cosmid sequences, and for recognizing multiple tandem domains in mosaic pro- teins. However, it does not have the ability to recognize fragments of matches, such as partial Alus; use `#hmmfs` for detecting partial as well as complete matches. The output has type: `{(#hmmfile: string, #hits: { (#bitsscore: num, #start: num, #end: num, #alignment: string)})}`.

  Note the differences between the different HMM searching programs. `#hmms` looks for a global alignment of HMM to sequences (Needleman/Wunsch style); overhangs of unmatches sequence are not permitted, and the full model must be matched. `#hmmls` looks for one or more local alignments of the full model to a subsequence of each database sequence; unmatches sequence overhangs are allowed. `#hmmsw` looks for the best fragmentary match of a subsequence to part of the model (Smith/Waterman style). `#hmmfs` looks for multiple non-overlapping matches of subsequences to parts of the model (modified multiple-hit Smith/Waterman). `#hmms` is useful for scoring or detecting whole proteins with complete models; `#hmmls` is useful for scoring or detecting intact domains in protein sequences or complete repeats in nucleic acid sequences; `#hmmsw` is useful for general protein database searching, allowing for possible incomplete matches to a model; `#hmmfs` is useful for general nucleic acid database searching or for scoring/detecting domains in multidomain proteins, allowing for multiple non-overlapping matches per sequence.

- `process <#hmmsw: "AFGIDLGTT...LGQTAGE" > using hmmer;`

  This kind of request finds the single best match of a subsequence to a fragment of the given model; ie., the program does Smith/Waterman-style searching. It is designed to be particularly useful for protein database searches with models of either whole proteins or protein domains. The output type is `string -> {(#hmmfile: string, #hits: { (#score: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num, #alignment: string)})}`.

- `process <#hmmfs: "AFGIDLGTT...LGQTAGE" > using hmmer;`

  This request finds an optimal set of non-overlapping multiple hits to subsequences and to fragments of the model are allowed for each database sequence—ie., the program does a multiple-hit variant of Smith/Waterman searching. It is designed to be particularly useful for detecting repeated motifs in long sequences. It has been used success- fully for Alu detection in human cosmid sequences, and for recognizing multiple tandem domains in mosaic proteins. The output has type `{(#hmmfile: string, #hits: { (#score: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num, #alignment: string)})}`.

- `process <#hmmall: "AFGIDLGTT...LGQTAGE" > using hmmer;`

  This kind of request combines all of the above kind of searches. The output looks like

  ```
  {<#hmms: (#hmmfile: "HSP70.HMM", #hits: {})>,
   <#hmmls: (#hmmfile: "HSP70.HMM", #hits: {})>,
   <#hmmsw: (#hmmfile: "HSP70.HMM",
            #hits: {(#score: 12.75,
                     #seqstart: 0, #seqend: 13,
                     #hmmstart: 0, #hmmend: 13,
                     #alignment: "VIGIDLGTTNSCVA
  ++GIDLGTT   ++
  AFGIDLGTTXXXLG")})>,
   <#hmmfs: (#hmmfile: "HSP70.HMM",
            #hits: {(#score: 7.11,
                     #seqstart: 0, #seqend: 13,
                     #hmmstart: 0, #hmmend: 13,
                     #alignment: "VIGIDLGTTNSCVA
  ++GIDLGTT   ++
  AFGIDLGTTXXXLG")})>}
  ```

  where the variant tag in each item indicates which search program is used.

- `process <#hmme-mostlikely: () > using hmmer;`

This request creates a set of likely sequences according to statistics of the given HMM (HSP70). This is sometimes useful for comparing different HMMs, or just spot-checking that an HMM contains reasonable numbers (i.e., a globin HMM should pro- duce sequences that look like globins). Only the most likely (ie. most probably) sequence is return. The output has type `{(#score: num, #sequence: string)}`.

  - `process <#hmme-bestscore: () > using hmmer;`

    Similar to `#hmme-mostlikely`. However, this time the highest scoring sequence is returned. Scoring uses a log-odds system, the probability of the sequence given the model divided by the probability of the sequence given a simple random sequence model. The output has type `{(#score: num, #sequence: string)}`.

  - `process <#hmme: (#number: 5 )> using hmmer;`

    Similar to `#hmme-mostlikely`. However, this time the specified number of sequences (5) is generated. The type of the output is `{(#name: string, #sequence: string)}`.

- `hmm-localtraining: (string, string, {(#uid: num, #accession: string, #title: string, #seq: string; '3) }) -> string`

  Create a HMM from a given set of sequences. The sequences may contain RNA, DNA, or amino acid; but please do not mix them! It works by iteratively improving a new multiple sequence alignment calculated using the model, then a new model given that alignment. A simulated annealing protocol is used to avoid bad local minima in the iterative (expectation maximi- zation) training procedure.

  The normal way to invoke this function is: `writefile` SEQUENCES `to` FILE-NAME `using hmm-localtraining`.

B.19 FASTA-Formated IO

FASTA is probably the most popular format for exchange protein sequences. Basically, each record in the file consists of a title line and several sequence lines. The title line must start with the > symbol. The sequence line must not be too long, 60 or so characters per line is about right. Here is how a FASTA file looks:

```
>d1cbn__ 7.11.1.1.1 Crambin [abyssinian cabbage (Crambe abyssinica)]
TTCCPSIVARSNFNVCRLPGTPEALCATYTGCIIIPGATCPGDYAN
>d2erl__ 1.10.1.1.1 ER-1 [Euplotes raikovi]
DACEQAAIQCVESACESLCTEGEDRTGCYMYIYSNCPPYV
>d8rxna_ 7.34.3.1.1 Rubredoxin [Desulfovibrio vulgaris]
MKKYVCTVCGYEYDPAEGDPDNGVKPGTSFDDLPADWVCPVCGAPKSEFEAA
>d1lkka_ 4.47.1.1.1 p56-lck tyrosine kinase [human (Homo sapiens)]
LEPEPWFFKNLSRKDAERQLLAPGNTHGSFLIRESESTAGSFSLSVRDFDQNQGEVVKHY
KIRNLDNGGFYISPRITFPGLHELVRHYTNASDGLCTRLSRPCQT
```

Since this format is so popular, we implemented drivers to input and output in this format. The interface is in module FASTAIO. Here are the available operations:

- `fasta-out: (#1: string, #2: string, #3: {(#title: string, #seq: string; ''3)}) -> string`

  This one writes a set of sequences out in FASTA format. Here is an example:

  ```
  writefile {(#title: "Rubredoxin", #seq: "MKKYVCTVCGYEYDPAEG..."),
             (#title: "tyrosine kinase", #seq: "LEPEPWFFKNLSRKDAERQL...")}
  to "/tmp/myfastafile" using fasta-out;
  ```

- `fasta-in: string -> {(#id: num, #title: string, #seq: string)}`

  This one reads a FASTA file in. The `#id` field corresponds to the sequence number, generated automatically by the system. Here is an example:

  ```
  readfile myfasta from "/tmp/myfastafile" using fasta-in;
  ```

- `fasta-slow: (#file: string, #burst: num, #sleep: num) -> {(#id: num, #title: string, #seq: string)}`

  This one also reads a FASTA file, but in burst mode. The `#file` field specifies the file. The `#burst` field specifies the burst size. The `#sleep` field specifies the number of seconds to sleep between bursts. Use this one instead of `fasta-in` if you need to read a very big FASTA file and you are only able to process its records very slowly.

B.20 Kenta Nakai's PSORT II

PSORT II is a long-awaited new version of PSORT, a program to predict the subcellular localization sites of proteins from their amino acid sequences. The algorithm is a very simple one, the k-nearest neighbors classifier, as described in [16].

Wang Jiren wrote a driver for this software. The interface to his driver is in module LOCALPSORT. It provides one function:

- `localpsort: string -> { ''3 }`

  You just apply it to your protein sequence to obtain cellular localization information. Here is an example:

  ```
  process "PVIFKASYHGLYYIITLVVVNGNV..." using localpsort;
  ```

  Here is an example output:

  ```
  {<#PSG: (#N-region: (#length: 5, #pos-chg: 1, #neg-chg: 0),
           #H-region: (#length: 21, #peak-value: 9.06),
           #score: 4.66, #misc: "")>,
   <#GvH: (#threshold: ~2.1, #score: ~5.98, #cleavagefrom: 43, #cleavageto: 44,
           #misc: "Seems to have no N-terminal signal peptide")>,
  ```

```
 <#ALOM: (#position: 1, #threshold: 0.5, #number: 1,
          #INTEGRAL: {(#likelihood: ~12.26, #from: 748, #to: 764)},
          #PERIPHERAL: (#likelihood: 0.58, #at: 10),
          #score: ~12.26, #misc: "Number of TMS(s) for threshold 0.5: 1")>,
 <#MTOP: (#centerposition: 755, #difference: 4.5, #c-value: 3.5,
          #n-value: ~1.0, #misc: "C > N: C-terminal side will be inside
Caution: Inconsistent mtop result with signal peptide
 membrane topology: type 1b (cytoplasmic tail 748 to 1071)")>,
 <#MITDISC: (#R-content: 1,
             #HydMoment: {(#value: 75, #Hyd-Moment: 6.34),
                          (#value: 95, #Hyd-Moment: 6.64)},
             #Gcontent: 2, #DEcontent: 1, #STcontent: 10,
             #Score: ~1.92, #misc: "")>,
 <#Gavel: (#misc: "prediction of cleavage sites for mitochondrial preseq
 R-2 motif at 40 SRS|IT")>,
 <#NUCDISC: (#misc: "discrimination of nuclear localization signals
 pat4: RKKH (3) at 765
 pat4: KKRK (5) at 830
 pat7: none
 bipartite: none
 content of basic residues: 9.2%
 NLS Score: 0.03")>,
 <#KDEL: (#misc: "ER retention motif in the C-terminus: none
ER Membrane Retention Signals:
 KKXX-like motif in the C-terminus: KKKQ")>,
 <#SKL: (#misc: "peroxisomal targeting signal in the C-terminus: none")>,
 <#PTS2: (#misc: "2nd peroxisomal targeting signal: none")>,
 <#VAC: (#misc: "possible vacuolar targeting motif: none
RNA-binding motif: none
Actinin-type actin-binding motif:
 type 1: none
 type 2: none")>,
 <#NMYR: (#misc: "N-myristoylation pattern : none
Prenylation motif: none")>,
 <#memYQRL: (#misc: "transport motif from cell surface to Golgi: none
Tyrosines in the tail: too long tail
Dileucine motif in the tail: found
 LL at 752
 LL at 756
 LL at 757
 LL at 806
checking 63 PROSITE DNA binding motifs: none
checking 71 PROSITE ribosomal protein motifs: none
checking 33 PROSITE prokaryotic DNA binding motifs: none")>,
 <#summary: {(#percent: 32.0, #location: "cytoplasmic"),
```

```
                 (#percent: 24.0, #location: "mitochondrial"),
                 (#percent: 20.0, #location: "nuclear"),
                 (#percent: 8.0, #location: "plasma membrane"),
                 (#percent: 8.0, #location: "endoplasmic reticulum"),
                 (#percent: 4.0, #location: "peroxisomal"),
                 (#percent: 4.0, #location: "vesicles of secretory system")}>}
```

B.21 NCBI Web Entrez

NCBI also makes its Entrez system available via the World Wide Web. The URL is

`http://www.ncbi.nlm.nih.gov/Entrez/protein.html`.

We have a set of driver routines for "Web" Entrez. These routines are implemented by various people (Chen Jing, Wang Jiren, Oliver Wu, Lin Kui, and myself.) The interface to these driver routines is in the Kleisli module WEBENTREZ.

- `aa-get-count-general : string -> { num }`

  Given a specification, returns a singleton set containing the count of Entrez records satisfying the specification. Basically a specification is just a Boolean combination of terms of the form `Key[Field]`. The allowed Boolean combinators are `AND`, `OR`, and `BUTNOT`. The allowed `Fields` are `Title Word`, `All Fields` (this is the default), `Accession`, `Author Name`, `Publication Date`, `EC/RN Number`, `Gene Name`, `Journal Name`, `Keyword`, `Modification date`, `Organism`, `Properties`, `Protein Name`, and `Text Word`.

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

  Given the uid of a protein sequence in Entrez, return its feature table.

- `aa-get-feature-by-uid': string -> {(#uid: num, #accession: string, #title: string, #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])})}`

  Same as `aa-get-feature-by-uid`, but you can use accession numbers also.

- `aa-get-feature2-by-uid: num -> {(#uid: num, #accession: string, #title: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])})}`

  Similar to `aa-get-feature-by-uid`, but returns database cross references as well.

- `aa-get-feature2-by-uid': string -> {(#uid: num, #accession: string, #title: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])`

  Similar to `aa-get-feature-by-uid'`, but returns database cross references as well.

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

  Same as `aa-get-feature-by-uid`, but you use Entrez specification as in `aa-get-count-general` instead of uid to specify the feature tables you want.

- `aa-get-feature2-general: string -> {(#uid: num, #accession: string, #title: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])`

  Similar to `aa-get-feature-general`, but returns database corss references as well.

- `aa-get-seqfeat-by-uid: num -> {(#uid: num, #title: string, #accession: string, #common: string, #organism: (#genus: string, #species: string, #lineage: [string]), #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])}, #sequence: string)}`

  Given the uid of a protein sequence in Entrez, return its feature table, lineage, and sequence.

- `aa-get-seqfeat-by-uid': string -> {(#uid: num, #title: string, #accession: string, #common: string, #organism: (#genus: string, #species: string, #lineage: [string]), #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])}, #sequence: string)}`

  Same as `aa-get-seqfeat-by-uid`, but you can also use accession numbers as input.

- `aa-get-seqfeat2-by-uid: num -> {(#uid: num, #title: string, #accession: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #common: string, #organism: (#genus: string, #species: string, #lineage: [string]), #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])}, #sequence: string)}`

  Similar to `aa-get-seqfeat-by-uid`, but returns database cross references as well.

  `aa-get-seqfeat2-by-uid': string -> {(#uid: num, #title: string, #accession: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #common: string, #organism: (#genus: string, #species: string, #lineage: [string]), #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)] #sequence: string)}`

  Similar to `aa-get-seqfeat-by-uid'`, but returns database cross references as well.

- `aa-get-seqfeat-general : string -> {(#uid: num, #title: string, #accession: string, #common: string, #organism: (#genus: string, #species: string, #lineage: [string]), #feature: {(#name: string, #start: num, #end: num, #anno: [(#anno_name: string, #descr: string)])}, #sequence: string)}`

  Same as `aa-get-seqfeat-by-uid`, but you use Entrez specification like `"metabotropic`AND+receptor”+ as input.

- `aa-get-seqfeat2-general: string {(#uid: num, #title: string, #accession: string, #dbsource: [(#db: string, #anno: string, #detail: [(#name: string, #value: string)])], #common: string,`

`#organism: (#genus: string, #species: string, #lineage: [ string]), #feature: {(#name: string, #start: num, #end: num, #anno: [( #anno_name: string, #descr: string)])}, #sequence: string)}`

Similar to `aa-get-seqfeat-general`, but returns database cross references as well.

- `aa-get-homolog-summary: num -> {(#uid: num, #accession: string, #name: string, #title: string, #weight: num)}`

  Given the uid of a protein sequence in Entrez, return summaries of its homologs.

- `aa-get-homolog-uid: num -> {num }`

  Given the uid of a protein sequence in Entrez, return uids of its homologs.

- `aa-get-lineage-by-uid: num -> {(#uid: num, #organism:string, #taxon:num, #lineage:[string])}`

  Given the uid of a protein sequence in Entrez, return its lineage.

- `aa-get-lineage-by-uid': string -> {(#uid: num, #organism:string, #taxon:num, #lineage:[string])}`

  Same as `aa-get-lineage-by-uid`, but you can use accession number, locus name, or other forms of unique identifiers accepted by Entrez.

- `aa-get-lineage-general: string -> {(#uid: num, #organism:string, #taxon:num, #lineage:[string])}`

  Same as `aa-get-lineage-by-uid`, but you use Entrez specification to specify the lineages you want instead of uid.

- `aa-get-ml-abstract: num -> {(#authors: string, #address: string, #title: string, #abstract: string, #journal: string)}`

  Given the uid of a protein sequence in Entrez, return abstracts of Medline articles mentioning it.

- `aa-get-ml-summary: num -> {(#muid: num, #author: string, #title: string, #journal: string)}`

  Given the uid of a protein sequence in Entrez, return summaries of Medline articles mentioning it.

- `aa-get-ml-uid: num -> {num}`

  Given the uid of a protein sequence in Entrez, return muids of Medline articles mentioning it. This one is different from `aa-get-muid-by-uid` as it does not inspect GenPept report. Note that due to the update policy of NCBI Entrez, neither of these two functions is a superset of the other.

- `aa-get-muid-by-uid: num -> {(#uid: num, #muid: {num}}`

  Given the uid of a protein sequence in Entrez, return muids of Medline articles mentioning it. This one is different from `aa-get-ml-uid` as it inspects the GenPept report for MedLine references.

- `aa-get-muid-by-uid': string -> {(#uid: num, #muid: {num}}`

  Like `aa-get-muid-by-uid`, but the uid can be given as a string.

- `aa-get-muid-general: string -> {(#uid: num, #muid: {num}}`

  Same as `aa-get-muid-by-uid` but you use Entrez specification to specify what you want instead of uid.

- `aa-get-n-summary-general: (#key: string, #count: num; '1) -> { (#uid: num, #accession: string, #name: string, #title: string) }`

  Given the specification in the `#key` field, return up to $n < 100$ summaries of matching Entrez protein records. n is given in the `#count` field.

- `aa-get-na-summary: { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given the uid of a protein sequence in Entrez, return summaries of DNA sequences linked to it.

- `aa-get-na-uid: num -> {num}`

  Given the uid of a protein sequence in Entrez, return uids of DNA sequences linked to it.

- `aa-get-report-by-uid: num -> {(#uid: num, #report: string)}`

  Given the uid of a protein sequence in Entrez, return the text of its full GenPept report.

- `aa-get-report-by-uid': string -> {(#uid: num, #report: string)}`

  Same as `aa-get-report-by-uid`, but you can use accession number, locus name, or any other form of unique identifiers accepted by Entrez.

- `aa-get-report-general: string -> {(#uid: num, #report: string)}`

  Same as `aa-get-report-by-uid`, but you can use Entrez specification instead of uid.

- `aa-get-seq-by-uid: num -> {(#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Given the uid of a protein sequence in Entrez, return the sequence.

- `aa-get-seq-by-uid': string -> {(#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Same as `aa-get-seq-by-uid`, but you can use accession number, locus name, or any other form of unique identifiers accepted by Entrez.

- `aa-get-seq-general: string -> {(#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Given a specification, return several hundred protein sequences whose description satisfies it.

- `aa-get-summary-by-uid: num -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given the uid of a protein sequence in Entrez, return its summary.

- `aa-get-summary-by-uid': string -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Same as `aa-get-summary-by-uid`, but you can use accession number, locus name, or any other form of unique identifiers accepted by Entrez.

- `aa-get-summary-general: string -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given a specification, return several hundred protein summaries whose description satisfies it.

- `aa-get-uid-general: string -> {num}`

  Given a specification, return uids of all proteins satisfying it.

- `ml-get-abstract-by-uid: num -> {(#muid: num, #authors: string, #address: string, #title: string, #abstract: string, #journal: string)}`

  Given the muid of a Medline article, return its abstract.

- `ml-get-abstract-by-uid': string -> {(#muid: num, #authors: string, #address: string, #title: string, #abstract: string, #journal: string)}`

  Given the muid (as a string) of a Medline article, return its abstract.

- `ml-get-abstract-general: string -> {(#muid: num, #authors: string, #address: string, #title: string, #abstract: string, #journal: string)}`

  Given Entrez specification of Medline articles, return their abstracts.

- `ml-get-summary-by-uid: num -> {(#muid: num, #author: string, #title: string, #journal: string)}`

  Given the muid of a Medline article, return its summary.

- `ml-get-summary-by-uid': string -> {(#muid: num, #author: string, #title: string, #journal: string)}`

  Given the muid (as a string) of a Medline article, return its summary.

- `ml-get-summary-general: string -> {(#muid: num, #author: string, #title: string, #journal: string)}`

  Given the Entrez specification Medline articles, return their summaries.

- `ml-get-aa-uid: num -> {num}`

  Given the muid of a Medline article, return uids of proteins mentioned by it.

- `ml-get-count-general: string -> { num}`

  Given a Medline specification, count the number of articles satisfying it.

- `ml-get-homology-by-uid: num -> {num}`

  Given the muid of a Medline article, return muids of related Medline articles.

- `ml-get-homology-general: string -> {num}`

  Use general Entrez specification to get muids of related Medline articles.

- `ml-get-na-uid: num -> { num}`

  Given the muid of a Medline article, return uids of DNA sequences mentioned by it.

- `ml-get-omim-by-uid: num -> { num}`

  Given the muid of a Medline article, return the identifier of related OMIM record. The OMIM record can be accessed via WWW using the URL `http://www.ncbi.nlm.nih.gov/htbin-post/Omim/dispmim?60` if 600185 is your OMIM identifier.

- `ml-get-omim-general: string -> {num}`

  Use general Entrez specification to get OMIM identifiers.

- `ml-get-uid-general: string -> {num}`

  Given a Medline specification, return the muids of all Medline articles satisfying it.

- `na-get-aa-summary: num -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given the uid of a DNA sequence in Entrez, return summaries of protein sequences linked to it.

- `na-get-feature-by-uid: num ->{(#uid: num,#title: string,#accession: string,#feature: {(#name: string,#continuous: bool,#position: [(#accn: string,#start: num,#end: num, #negative: bool)], #anno: [ (#anno_name: string, #descr: string)])})}`

  Given uid of DNA record, return its feature table. The `#continuous` field of a feature is true if the positions of that feature are contiguous. The `#negative` field of a position is true if that position must be reverse-complemented.

- `na-get-feature-by-uid': string ->{(#uid: num, #title: string, #accession: string, #feature:{(#name: string,#continuous: bool,#position: [(#accn: string,#start: num, #end: num, #negative: bool)], #anno: [ (#anno_name: string, #descr: string)])})}`

  Same as `na-get-feature-by-uid`, but you can use accession numbers, locus names, etc.

- `na-get-feature-general: string ->{(#uid: num, #title: string, #accession: string, #feature:{(#name: string,#continuous: bool,#position: [(#accn: string,#start: num, #end: num, #negative: bool)], #anno: [ (#anno_name: string, #descr: string)])})}`

  Use general Entrez specification to get feature tables.

- `na-get-aa-uid: num -> {num}`

  Given the uid of a DNA sequence in Entrez, return uids of protein sequences linked to it.

- `na-get-count-general: string -> { num}`

  Given a specification, count the number of DNA sequences in Entrez satisfying it.

- `na-get-homolog-summary: num -> {(#uid: num, #accession: string, #name: string, #title: string, #weight: num)}`

  Given the uid of a DNA sequence in Entrez, return summaries of its homologs.

- `na-get-homolog-uid: num -> {num}`

  Given the uid of a DNA sequence in Entrez, return uids of its homologs.

- `na-get-lineage-by-uid: string -> {(#uid: num,#organism:string,#taxon:num,#lineage:[string])}`

  Given the uid of a DNA sequence in Entrez, return its lineage.

- `na-get-lineage-by-uid': string -> {(#uid: num,#organism:string,#taxon:num,#lineage:[string])`

  Same as `na-get-lineage-by-uid`, but you can use accession number, locus name, or other forms of identifiers accepted by Entrez.

- `na-get-lineage-general: string -> {(#uid: num, #organism:string, #taxon:num, #lineage:[string])`

  Same as `na-get-lineage-by-uid`, but you can use Entrez specification instead of uid.

- `na-get-ml-abstract: num -> {(#authors: string, #address: string, #title: string, #abstract: string, #journal: string)}`

  Given the uid of a DNA sequence in Entrez, return Medline abstracts linked to it.

- `na-get-ml-summary: num -> {(#muid: num, #author: string, #title: string, #journal: string)}`

  Given the uid of a DNA sequence in Entrez, return Medline summaries linked to it.

- `na-get-ml-uid: num -> {num}`

  Given the uid of a DNA sequence in Entrez, return muids of Medline articles linked to it.

- `na-get-n-summary-general: (#key: string, #count: num; '1) -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given the specification in the `#key` field, return up to $n < 100$ summaries of matching Entrez DNA records. n is given in the `#count` field.

- `na-get-report-by-uid: num -> {(#uid: num, #report: string)}`

  Given the uid of a DNA sequence in Entrez, return its full GenBank report.

- `na-get-report-by-uid': string -> {(#uid: num, #report: string)}`

  Same as `na-get-report-by-uid`, but you can use accession number, locus name, or other forms of identifiers accepted by Entrez.

- `na-get-report-general: string -> {(#uid: num, #report: string)}`

  Same as `na-get-report-by-uid`, but you can use Entrez specification instead of uid.

- `na-get-seq-by-uid: num -> { (#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Given the uid of a DNA sequence in Entrez, return the sequence.

- `na-get-seq-by-uid': string -> { (#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Same as `na-get-seq-by-uid`, but you can use accession number, locus name, or other forms of identifiers accepted by Entrez.

- `na-get-seq-general: string -> { (#uid: num, #accession: string, #title: string, #name: string, #sequence: string)}`

  Given a specification, return several hundred summaries of DNA sequences satisfying it.

- `na-get-summary-by-uid: num -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Given the uid of a DNA sequence in Entrez, return its summary.

- `na-get-summary-by-uid': string -> { (#uid: num, #accession: string, #name: string, #title: string)}`

  Same as `na-get-summary-by-uid`, but you can use accession number, locus name, or other forms of identifiers accepted by Entrez.

- `na-get-summary-general: string -> {(#uid:num, #accession: string, #name: string, #title: string)}`

  Given a specification, return summaries of several hundred summaries of DNA sequences satisfying it.

- `na-get-uid-general: string -> { num }`

  Given a specification, return uids of all DNA sequences satisfying it.

B.22 Louxin's MinAlig

Zhang Louxin wrote a pairwise alignment program MinAlig for us. The interface to this program is the Kleisli module MINALIG. This interface provides one primitive.

- `minalig-doit: (#1: string, #2: string) -> { (#Matching-Percentage: num, #Alignment-Score: num, #Match-Char: num, #Mismatch: num, #Indel: num, #Seq1: string, #Seq2: string)}`

  It aligns two sequences like this:

  ```
  minalig-doit ("ABCEGHY", "ABCCCHYY");
  Result:
  {(#Matching-Percentage: 71.428571, #Alignment-Score: 62.5,
    #Match-Char: 5, #Mismatch: 2, #Indel: 1,
    #Seq1: "ABCEGH-Y",
    #Seq2: "ABCCCHYY")}
  ```

B.23 ClustalW

ClustalW [36] is a very good software for multiple alignment of proteins. The interface to it is the Kleisli module CLUSTALWSVR. The module provides a server call.

- `clustalw: [ (#uid: num, #accession: string, #seq: string; ''3) ] -> { string }`

  You supply the proteins you want to multiple align in a list. The output produced is a set containing a single string, representing the alignment. Here is an example:

  ```
  process
    [(#uid: 1, #accession: "A", #seq: "AAAAQQQRRRRRYYYYYII"),
     (#uid: 4, #accession: "B", #seq: "ABBAAQQQRRRYYYIILLLLL"),
  ```

```
   (#uid: 3, #accession: "C", #seq: "RRBAAAAARRRRYYYYLII") ]
using clustalw;
Result:
{"0_1|A           -AAAAQQQRRRRRYYYYII----
1_4|B           -ABBAAQQQRRR-YYYIILLLLL
2_3|C           RRBAAAAARRRRYYYYLII----
                    *   .*** *** *.
"}
```

B.24 HTML Table Output

Since output from CPL is often shown on the web, it would be nice to able to format HTML tables. We provide a output server for doing that in the Kleisli module TABLEFORMAT. Here is the interface.

- `set-tableout: (#name: string, #border: bool, #spec: ''1) -> string`

  We explain it via an example.

  ```
  set-tableout (
    #name: "my-set-out",
    #border: true,
    #spec: (
       #employee: (
          #headtext: "Employee",
          #headcolour: "#AA0000",
          #cellcolour: "#AAAA00"),
      #salary:  (
          #headtext: "Monthly Salary",
          #headcolour: "#AA0000",
          #cellcolour: "#AAAA00")));
  ```

  The above creates a output server `my-set-out` for printing sets in tabular form. The `#border` field indicates if you want borders for your table. The `#spec` field is a record whose fields correspond one-to-one to the columns of the table. For each column, you specify the words to be used for its heading (`#headtext`), the background colour for that heading (`#headcolour`), and the background colour for that column (`#cellcolour`). You use it like this:

  ```
  process
    { (#employee: "John", #salary: 1000),
      (#employee: "Peter", #salary: 2000) }
  to "/tmp/my-output.html" using my-set-out;
  ```

  The set above will be written to the file `/tmp/my-output.html` which you can then view using Netscape. It looks like this, minus the colour:

Employee	Monthly Salary
John	1000
Peter	2000

If the file name is empty (ie., `""`), the output is written to the screen. Also, the fields does not need to be atomic, you can put any complex object in any field.

- `bag-tableout: (#name: string, #border: bool, #spec: ''1) -> string`

  Similar to `set-tableout`, but works for bags.

- `list-tableout: (#name: string, #border: bool, #spec: ''1) -> string`

  Similar to `set-tableout`, but works for lists.

B.25 HTML Nested Table Output

We also have a Kleisli module NESTEDTABLEFORMAT that can print "nested" table by using the ROW-SPAN feature of HTML. Here is the interface:

- `list-groupout: (#name: string, #border: bool, #group: ''2, #nongroup: ''1) -> string`

  We explain using an example.

```
list-groupout (
  #name: "my-table-out",
  #border: true,
  #group: (
     #emp-id: (
        #headtext: "Id",
        #headcolour: "#AA0000",
        #cellcolour: "#AAAA00"),
    #emp-name: (
        #headtext: "Name",
        #headcolour: "#AA0000",
        #cellcolour: "#AAAA00")),
  #nongroup: (
    #phone: (
        #headtext: "Phone",
        #headcolour: "#AA0000",
        #cellcolour: "#AAAA00")));
```

  The above creates a output server `my-table-out` that prints lists. If the `#group` fields of consecutive items on the list are equal, then the `#nongroup` fields are printed as a subtable. To illustrate:

```
writefile
  [ (#group: (#emp-id: 1, #emp-name: "John"), #nongroup: (#phone: "1234")),
    (#group: (#emp-id: 1, #emp-name: "John"), #nongroup: (#phone: "6678")),
    (#group: (#emp-id: 2, #emp-name: "Peter"), #nongroup: (#phone: "1345")),
    (#group: (#emp-id: 1, #emp-name: "John"), #nongroup: (#phone: "6345"))]
to "/tmp/my-output.html" using my-table-out;
```

gives an output that looks like this in a web browser:

Id	Name	Phone
1	John	1234
		6678
2	Peter	1345
1	John	6345

If the file name is empty (ie., `""`), the output is written to the screen. The fields are not required to be atomic.

Note that the third phone number of John is not grouped with his first two phone numbers, because they are interrupted by the phone number of Peter. But you can do a simple sort on the `#group` field if you really want the phone number of John to be printed together!

- `list-groupout': (#name: string, #border: bool, #group: ''1, #nongroup: ''2) -> string`

  Same as `list-groupout`, but creates a output server that writes the nested table out as a string.

- `list-groupout3: (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3) -> string`

  Same as `list-groupout`, but is for nested table of depth 3.

- `list-groupout3': (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3) -> string`

  Same as `list-groupout'`, but is for nested table of depth 3.

- `list-groupout4: (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4) -> string`

  Same as `list-groupout`, but is for nested table of depth 4.

- `list-groupout4': (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4) -> string`

  Same as `list-groupout'`, but is for nested table of depth 4.

- `list-groupout5: (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5) -> string`

  Same as `list-groupout`, but is for nested table of depth 5.

- `list-groupout5': (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5) -> string`

  Same as `list-groupout'`, but is for nested table of depth 5.

- `list-groupout6: (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5, #level6: ''6) -> string`

  Same as `list-groupout`, but is for nested table of depth 6.

- `list-groupout6': (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5, #level6: ''6) -> string`

  Same as `list-groupout'`, but is for nested table of depth 6.

- `list-groupout7: (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5, #level6: ''6, #level7: ''7) -> string`

  Same as `list-groupout`, but is for nested table of depth 7.

- `list-groupout7': (#name: string, #border: bool, #level1: ''1, #level2: ''2, #level3: ''3, #level4: ''4, #level5: ''5, #level6: ''6, #level7: ''7) -> string`

  Same as `list-groupout'`, but is for nested table of depth 7.

B.26 Indexed Files

Hwee Hwa's SeqIndex can also be used to build normal indexed files. This is implemented by the Kleisli module INDEXEDFILESVR. The interface is given below.

- `setindex-create: (#1: string, #2: string, #3: string) -> string`

  For creating an index on an existing file. Works like this:

  ```
  writefile { (#name: "John", #age: 10), (#name: "Peter", #age: 15)}
  to "/tmp/mysetfile" using machineout;
  writefile "/tmp/mysetfile" to "#name" using setindex-create;
  ```

- `bagindex-create: (#1: string, #2: string, #3: string) -> string`

  For creating an index on an existing file. Works like this:

  ```
  writefile {| (#name: "John", #age: 10), (#name: "Peter", #age: 15) |}
  to "/tmp/mybagfile" using machineout;
  writefile "/tmp/mybagfile" to "#name" using setindex-create;
  ```

- `listindex-create: (#1: string, #2: string, #3: string) -> string`

  For creating an index on an existing file. Works like this:

```
writefile [ (#name: "John", #age: 10), (#name: "Peter", #age: 15) ]
to "/tmp/mylistfile" using machineout;
writefile "/tmp/mylistfile" to "#name" using setindex-create;
```

- `setindex-access:(#name:string, #file:string, #key:string)->string`

  For accessing an index. Works like this:

  ```
  setindex-access (#name: "name2data", #file: "/tmp/mysetfile", #key: "#name");
  ```

  The above opens a connection `name2data` to the index on the `#name` field of the file `/tmp/mysetfile`. This connection accepts two kinds of requests. This first one has the form `<#all: ()>`, which simply returns everything in the file:

  ```
  process <#all: ()> using name2data;

  Result:
  {(#name: "John", #age: 10), (#name: "Peter", #age: 15)}
  ```

  The second one given a key (a name in this case), returns all corresponding records:

  ```
  process <#key: "John"> using name2data;

  Result:

  {(#name: "John", #age: 10)}
  ```

- `bagindex-access:(#name:string, #file:string, #key:string)->string`

  Ditto. But works on bags.

- `listindex-access:(#name:string, #file:string, #key:string)->string`

  Ditto. But works on lists.

B.27 ISREC ProfileScan Server

ISREC has a good profile scanning program pfscan2.0 [3]. The Kleisli module WEBPFSCAN implements driver to their web server at `http://ulrec3.unil.ch/software/PFSCAN_form.html`. The interface is given below.

- `webpfscan-add: (#name: string, #profile: string, #sensitivity: string) -> string`

  Opens a connection to ISREC ProfileScan server. `#name` is name of connection. `#profile` is the profile you want to use; acceptable values are "prosite" (PROSITE Profiles with pre-release), "patterns" (PROSITE patterns), "pfama_nat" (Pfam A bitscaled), "pfama_mod" (Pfam A NScores), "gribskov" (Gribskov collection Z-scores), and "all" (PROSITE + Pfam A Nscores). `#sensitivity` is the sensitivity level; acceptable values are "df" (default sensitivity), "sm" (significant match only), "im" (include weak match).

  Here is an example.

```
webpfscan-add (#name: "mypfscan", #profile: "prosite", #sensitivity: "df");
process "MY SEQUENCE" using mypfscan;

Result:

[(#normalize: 16.6865, #raw: 663, #start: 8, #end: 101,
  #prositeid: "PS50167", #title: "GLYC_TRANS",
  #description: "General Glycosyltransferase domain. ",
  #significant: true), ...]
```

B.28 Local Version of ISREC's ProfileScan 2.0

ISREC's Pfscan 2.0 [3] is a nice tool for finding domains. The Kleisli module LOCALPFSCAN is a driver to a local copy of this tool. The interface is given below.

- `localpfscan-add: #name: string, #profile: string) -> string`

  Opens a connection to a profile. `#name` is name of the connection. `#profile` is file name of the profile. The ones that come with the ISREC distribution are "pfama_mod.prf", "pfama_nat.prf", "prosite_all.prf", "prosite_preliminary.prf", "prosite_released.prf", "prosite_submitted.prf".

  Here is an example.

```
localpfscan-add (
  #name: "mypfscan",
  #profile: "/usr49/BIC/pft_profiles/prosite_all.prf");
process "YOUR PROTEIN SEQUENCE" using mypfscan";

Result:
{(#N-score: 7.1393995, #Begin: 244, #End: 277, #score: 164,
  #Prositeid: "PS50005", #Name: "TPR_REPEAT", #Description: "TPR repeat.",
  #Match: "YELLYHLALLERRQGDLDAAIDRYQAALQEPVDA"),
...}
```

B.29 Pedersen and Nielsen's NetStart

Pedersen and Nielsen [28] has a program called NetStart that predicts translation initiation sites in vertebrates and Arabidopsis. Their server is at `http://www.cbs.dtu.dk/services/NetStart/`. The Kleisli module NETSTART is a driver to their server. the interface is given below.

- `netstart-add: (#name: string, #organism: string) -> string`

  Opens a connection to Pedersen's NetStart server. `#name` is name of the connection. `#organism` specifies the organism; acceptable values are "vert" (vertebrates) and "at" (Arabidopsis).

  Here is an example.

```
netstart-add (#name: "ini-site", #organism: "at");
process "YOUR ARABIDOPSIS DNA SEQUENCE" using ini-site;

Result:

[(#position: 52, #score: 0.552, #significant: true),
 (#position: 56, #score: 0.276, #significant: false),
 (#position: 115, #score: 0.405, #significant: false),
 (#position: 121, #score: 0.611, #significant: true),
...]
```

B.30 Fukuda's Kex

Kex is a program for finding protein names in MEDLINE abstracts [13]. The Kleisli module KEXSVR is a driver for it. The interface is given below.

- `kex-doit: string -> {string}`

  Finds all the protein names in the input string. It does a pretty good job on MEDLINE abstracts. For example, on

```
process
  "p21cip1/waf1/sdi1 is a universal cyclin-Cdk kinase inhibitor
  that has two functional domains; one binds and inhibits cyclin-Cdk
  activity and the other binds PCNA and thereby inhibits elongation by
  DNA polymerase. When transiently expressed in hamster BHK21 cells
  we found that human p21 was able to cause cell cycle arrest in G1 phase;
  this arrest was counteracted by coexpression of E2F-1 or SV40
  large T antigen. To study the effect of p21 overexpression in vivo,
  BHK21 cell clones inducibly expressing human p21 (Tet-p21) driven
  by the tetracycline (Tet)-repressible promoter were established. The
  maximum induced p21 levels in the absence of Tet were estimated to
  be ten times that of endogenous hamster p21. As p21 levels rose
  following removal of Tet, p21-associated histone H1 kinase activity
  was increased and concomitantly cell growth and DNA synthesis were
  reduced. Tet-p21 BHK21 cells became arrested in G1 phase and lost
  colony forming ability irreversibly 2-4 days after removal of Tet.
  The induction of cyclin E- and cyclin A-associated kinase activities
  was diminished when G0-synchronized Tet-p21 BHK21 cells were
  serum stimulated in the absence of Tet. Increased binding of p21 to
  PCNA and cyclin D1-Cdk4 was detected in induced cells.
  Overexpression of p21 led to cell death in BHK21 cells at 39.5 degrees C
  within 4 days."
using kex-doit;
```

produces

```
{"universal cyclin-Cdk kinase inhibitor",
 "PCNA", "p21", "G1 phase", "E2F-1", "SV40",
 "p21 ( Tet-p21 )", "tetracycline ( Tet )",
 "Tet", "endogenous hamster p21",
 "p21-associated histone H1 kinase",
 "cyclin E", "cyclin", "G0-synchronized Tet-p21",
 "cyclin D1-Cdk4"}
```

B.31 TIGR

This subsection is written by Lin Kui.

TIGR Database(TDB) is a collection of curated databases containing DNA and protein sequences, gene expression, cellular role, protein family, and taxonomic data for microbes, plants and humans. See `http://www.tigr.org.` There are many databases in TDB, so far, we only have the following drivers to connect to the HGI–Human Gene Index, and EGAD–Expressed Gene Anatomy Database, etc. HGI contains human EST sequences sequenced at TIGR as well as human ESTs from GenBank's dbEST database. It integrates research results from international human gene research projects, including research at TIGR. The Index is an outgrowth of TIGR's Human cDNA Database (HCD) and Expressed Gene Anatomy Database (EGAD) projects. EGAD was constructed by extraction and curation of sequences from GenBank to create a non-redundant set of human transcript (HT) sequences. In some cases, HTs were created by splicing together distinct GenBank accessions for each exon in those transcripts, or by splicing exons from a genomic sequence. Each HT sequence is linked to a human gene (HG). An HG may have more than one HT due to alternative splicing. HG and HT names have been curated for consistent nomenclature. Cellular roles have been assigned to each HT. Tissue distribution information is available for most HTs. Extensive expression information, including positions of overlapping Expressed Sequence Tags (ESTs), is available in HT reports (for most HTs) through links to TIGR's Human Gene Index (HGI). Clones for HTs may also be available through the TIGR/ATCC Special Collection.

- `tigr-humgen-get-id-by-name: string -> {(#est-number: <#est: string, #thc: string, #ht: string>, #putative-id: string)}`

  Given a gene name returns a set `{(#est-number: <#est: string, #thc: string, #ht: string>, #putative-id: string)}`. `#est` means this is a piece of Expressed Sequence Tags. `#thc` means this is one of Tentative Human Consensus sequences, which are assemblies of human ESTs. `#ht` means this is one of mature Human Transcript sequences, which are part of TIGR's Expresed Gene Anatomy Database(EGAD).

  Note that each HT sequence is linked to a human gene(HG). An HG may have more than one HT due to alternative splicing. When you have this result, you can use it to get corresponding reports by using drivers: `tigr-get-ht-by-id`, `tigr-humgen-get-thc-by-id` and `tigr-humgen-get-est-by-id`.

  Example:

```
tigr-humgen-get-id-by-name "brca1";
Results:
{( #est-number: <#est: "AA248220">,  #putative-id: "BRCA1-associated" ),
 ( #est-number: <#est: "AA261880">,  #putative-id: "BRCA1-associated" ),
 ( #est-number: <#thc: "THC154499">, #putative-id: "breast/ovarian ..." )}
```

- `tigr-humgen-get-thc-by-id: string -> {(#id: string, #putative-id: string, #seq: string, #overlap-with: {(#est-id: string, #est-source: string, #gb: string, #atcc: string, #left: num, #right: num, #library: string)})}`

  Given a THC number, returns its associated information. Tentative Human Consensus sequences are created by assembling ESTs into virtual transcripts. In some cases, THCs contain full or partial cDNA sequences (HTs) obtained by classical methods. THCs contain information on the source library and abundance of ESTs and in many cases represent full-length transcripts. Alternative splice forms are built into separate THCs. Hence, THCs are actual assemblies, with a consensus sequence, and not simply clusters of overlapping sequences.

  Example:

```
tigr-humgen-get-thc-by-id ""THC101183";
Result:
{(#id: "THC101183", #putative-id: "Not assigned",
  #seq: "ATTGACTTCCTGCAGAGAGTCAGCCCCACCAATA...",
  #Overlap-with: {
    (#est-id: "", #est-source: "",  #gb: "F08725", #atcc: "",
     #left: 1, #right: 360, #library: "Infant brain cDNA"),
    (#est-id: "yq67f08.r1", #est-source: "WashU/Merck",
     #gb: "R98879", #atcc: "447616", #left: 8, #right: 319,
     #library: "Fetal liver"),
    (#est-id: "yg17g02.r1", #est-source: "WashU/Merck",
     #gb: "R18769", #atcc: "358443", #left: 75, #right: 459,
     #library: "Infant brain 1NIB, Soares")})}
```

- `tigr-humgen-get-est-by-id: string -> {(#id: string, #putative-id: string, #seq: string, #overlap-with: {(#est-id: string, #gb: string, #atcc: string, #clone-end: string, #cat: string, #thc: string, #opposite: string)})}`

  Given an EST number, returns its associated information. Expressed Sequence Tags are partial, single-pass sequences from either end of a cDNA clone. The EST strategy was developed to allow rapid identification of expressed genes by sequence analysis.

  Example:

```
tigr-humgen-get-est-by-id  "AA248220";
Result:
{(#id: "AA248220", #putative-id: "BRCA1-associated",
  #seq: "ATGAATCATCCTCAGCTAGCCACTGCTCAGTAAT...",
  #overlap-with: {(#est-id: "AA248220", #gb: "AA248220",
     #atcc: "", #clone-end: "Unknown", #cat: "LB77",
     #thc: "", #opposite: "" )})}
```

- `tigr-get-ht-by-id: string -> {(#gene-id: num, #sequence-id: num, #protein-id: num, #sequence-name: string, #genome: string, #taxon: string, #coding-length: num, #transcript-length: num, #expression: string, #derivation: {(#db: string, #accession: string, #title: string)}, #cdna-features: [(#feature: string, #5': num, #3': num)], #sequence-dna: string, #sequence-aa: string)}`

  Given an EGAD id (HTxxxx) returns its associated information. TIGR's Expressed Gene Anatomy Database (EGAD) contains a non-redundant set of nucleotide sequences that represent mature human transcripts (HTs). The HTs are curated for nomenclature and cellular roles, and links have been made to related accessions. Sequences were either loaded directly from GenBank (cDNAs) or were derived from genomic sequences when cDNAs were not available. Where available, 5' and 3' non-coding regions were included. Alternative splice forms of genes are explicitly represented.

  Example:

  ```
  tigr-get-ht-by-id "HT27473";
  Result:
  {(#gene-id: 26993, #sequence-id: 27473, #protein-id: 26152,
    #sequence-name: "antiproteinase 2, cytoplasmic", #genome: "nucleus",
    #taxon: "Homo sapiens", #coding-length: 1125,
    #transcript-length: 1325, #expression: "THC151617",
    #derivation: [
      (#db: "SP", #accession: "P50452", #title: "CYTOPLASMIC ..."),
      (#db: "GB", #accession: "L40377", #title: "Homo sapiens ...")],
    #cdna-features: {(#feature: "coding_seq", #5': 84, #3': 1208)},
    #sequence-dna: "AGCATCTACAAAGGAGGAATAGTCAAAGCAGCAGCGG...",
    #sequence-aa:  "MDDLCEANGTFAISLFKILGEEDNSRNVFFSPMSISS...")}
  ```

B.32 Jean-Michel Claverie's ORF Extraction

This subsection is prepared by Liu Hai.

We have modified Jean-Michel Claverie's ORF prediction program to output in Kleisli's Standard Exchange Format. The interface to this modified program is LOCALORF. It makes available the following primitive in CPL:

- `localorf-doit: (#seq: string, #min: num, #max: num) -> {(#begin: num, #end: num, #length: num, #seq: string)}`

  Extract all "open reading frames (orfs) from a nucleotide sequence within a give range, which normally is from 60 nt to 10000 nt. It returns a set of protein sequences of the format: `(#begin :num, #end: num, #length: num, #seq: string)` where `#begin` and `#end` represents the location in the given nucleotide sequence, `#length` is the length of this protein sequence, and `#seq` is the sequence itself.

B.33 Warren Gish's WU-BLAST2 (Extra Options)

A second interface to Warren Gish's WU-BLAST2 is in the module LOCALBLASTOPT. The functions provided by this interface are given below. The main difference of this interface from the first interface (see SubsectionB.10) is that you can specify all sorts of commandline options.

- `localblast-blastp-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Same as `localblast-blastp`. However, the `#option` argument let you specify some additional command-line parameters allowed by BLASTP of the WU-BLAST2 suite. For example, `"-filter seg"` sets the filter method to SEG, `"-filter` segxnu"+ sets the filter method to SEG and XNU.

- `localblast-blastn-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Ditto.

- `localblast-blastn-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Ditto.

- `localblast-blastx-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Ditto.

- `localblast-tblastx-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Ditto.

- `localblast-tblastn-detail-opt: (#name: string, #db: string, #option: string) -> string`

  Ditto.

B.34 Sean Eddy's HMMER 2.0

This subsection is written by Wang Jiren.

Profile hidden Markov models (profile HMMs) can be used to do sensitive database searching using statistical descriptions of a sequence family's consensus. HMMER 2.0 is an almost complete rewrite of the entire package, with improved sensitivity and a number of other features.

The interface for Sean Eddy's HMMER 2.0 driver is in the module HMMER2. It provides the functions listed below:

- `hmm2-local: (#name: string, #model: string, #ecutoff: num, #level: num) -> string`

  Establish connections to HMMER2. The number of connections is given by the `#level` field. The name of the connections is specified by the `#name` field. The field `#model` is the HMM file which you want to operate on and the field `#ecutoff` is the E-value cutoff for the per-sequence ranked hit list in searching a sequence database with a profile HMM or searching a single sequence against an HMM database. Hits with E-value better than (less than) this threshold will be shown.

  Below is an example for establishing a connection to HMMER2:

  ```
  hmm2-local(#name: "hmmer2",
             #model: "/tmp/rrm.hmm",
             #ecutoff: 10.0,
             #level: 2);
  ```

  After establishing the connection to HMMER2, you can operate on rrm.hmm by sending request to this connection in the following methods:

  - `process <#build: (#seqfile: {(#uid: num, #accession: string, #seq: string)}, #familyname: string, #alignstyle: string, #append: bool, #dna: bool)> using hmmer2;`

    This request builds a new profile HMM from a set of given sequences. The sequences may contain only nucleic acid sequences (either DNA or RNA) or only amino acid sequences. Set field `#dna` true if the sequences are nucleic acid sequences or false if they are amino acid sequences. The field `#familyname` represents the family name or specified name where the above sequences come from. When you build the model, you need to choose what kind of alignments you want to allow. There are four alignment styles in the request, namely `hmmls`, `hmmfs`, `hmms` and `hmmsw`. The option `hmmls` is used to search a sequence for local matches to an HMM, the option `hmmfs` is used to search for multiple fragments in long sequences (repeats or motifs), the option `hmms` is used to search a sequence for global matches to an HMM, and option `hmmsw` is used to search a sequence with an HMM, looking for local matches of part of the sequence against part of the HMM. Set field `#append` to true if you append built model to an existing model file specified by `#model` field in above creating a connection to HMMER2 example `hmm2-local(#name: "hmmer2", #model: "/tmp/rrm.hmm", #ecutoff: 10.0, #level: 2)`. Set field `#append` to false if you create a new model file.

    Here is an examples:

    ```
    process <#build: (#seqfile:
                       { (#uid: 1,
                          #accession: "A",
                          #seq: "SRLGSYQHDKILVPLKIEAIVSVIQTNQ"),
                         (#uid: 2,
                          #accession: "B",
                          #seq: "ELSRTTRGRHVHQAISMPKVGRSKGCFGRRLD")},
                     #familyname: "fn3",
                     #alignstyle: "hmmls",
                     #append: false,
                     #dna: false)> using hmmer2;
    ```

- `process <#calibrate: ()> using hmmer2;`

  This request is optional, but doing it will increase the sensitivity of your database search. It may take several minutes (or longer) to run.

- `process <#emit: (#number: num)> using hmmer2;`

  This request generates a number of sequences from an existing model file specified by `#model` field in a connection to HMMER2. This can be useful for various applications in which one needs a simulation of sequences consistent with a sequence family consensus. The field `#number` is the number of sequences to be generated. The output has type `{(#name: string, #seq: string)}`.

  Below is an example of this request and its output:

  ```
  process <#emit: (#number: 3)> using hmmer2;
  Result:
  {(#name: "seq1",
    #sequence: "NLQRSTRGQTMHDFILVPRLKRKVCVILRLQ"),
   (#name: "seq2",
    #sequence: "PGSKSPLPSHQDAIISVPAMTKAAGCFGQRVD"),
   (#name: "seq3",
    #sequence: "MHTADSPARVIRSAVNVQTRDD")}
  ```

- `process <#align: [(#uid: num, #accession: string, #seq: string)]> using hmmer2;`

  This request aligns the given sequences to the profile HMM and outputs a multiple sequence alignment. The output has the format `{(#uid: num, #seq: string)}`. The example of this request and its output are:

  ```
  process <#align: [ (#uid: 1,
                       #accession: "A",
                       #seq: "PGSKSPLPSHQDAIISVPAMTKAAGCFGQRVD"),
                      (#uid: 2,
                       #accession: "B",
                       #seq: "MHTADSPARVIRSAVNVQTRDD")]>
  using hmmer2;
  Result:
  {(#uid: 1,
    #seq: "pg..SKSPLPSHQDAIISVPAMTKAAGCFGQRVD"),
   (#uid: 2,
    #seq: "............MHTADSPARVIRSAVNVQTRDD")}
  ```

- `process <#search: string> using hmmer2;`

  This request is used to search signigicantly similar sequence matches against an existing model file specified by `#model` field in a connection to HMMER2. The output has the type `{(#hmmfile: string, #score: num, #E-value: num, #n: num, #hits: { (#score: num, #E-value: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num, #alignment: string)})}`.

  An example and its result are listed below:

  ```
  process <#search: "PGSKSPLPSHQDAIISVPAMTKAAGCFGQRVD"> using hmmer2;
  ```

```
Result:
{(#hmmfile: "/tmp/rrm.hmm",
  #score: 21.0,
  #E-value: 4.6E~7,
  #n: 1,
  #hits: {(#score: 21.0,
           #E-value: 4.6E~7,
           #seqstart: 2,
           #seqend: 31,
           #hmmstart: 0,
           #hmmend: 31,
           #alignment: "ELsRsrlGryvHdkIlvPkkirakvsvirrld
                          s+s l +++   I+vP   +a +++++r d
                        --SKSPLPSHQDAIISVPAMTKAAGCFGQRVD")})}
```

- `process <#pfam: string> using hmmer2;`

  This request compares a single sequence against all the HMMs in an existing model file specified by `#model` field in a connection to HMMER2 and looks for signigicantly similar sequence matches. The output has the type `{(#hmmfamily: string, #score: num, #E-value: num, #n: num, #hits: {(#score: num, #E-value: num, #seqstart: num, #seqend: num, #hmmstart: num, #hmmend: num, #alignment: string)})}`.

  The example of this request and its result are below:

```
process <#pfam: "PGSKSPLPSHQDAIISVPAMTKAAGCFGQRVD"> using hmmer2;
Result:
{(#hmmfamily: "fn3",
  #score: 21.0,
  #E-value: 0.027,
  #n: 1,
  #hits: {(#score: 21.0,
           #E-value: 0.027,
           #seqstart: 2,
           #seqend: 31,
           #hmmstart: 0,
           #hmmend: 31,
           #alignment: "ELsRsrlGryvHdkIlvPkkirakvsvirrld
                          s+s l +++   I+vP   +a +++++r d
                        --SKSPLPSHQDAIISVPAMTKAAGCFGQRVD")})}
```

- `hmm2-std: (#name: string, #ecutoff: num, #level: num) -> string`

  Same as `hmm2-local`, but uses the default file of HMM models from PFAM.

B.35 NCBI's BLAST 2.0

This subsection is written by Chen Jing.

The NCBI's BLAST 2.0 server has significant differences from their previous web BLAST server. These include significant performance enhancements, the addition of 'gapping' routines and position-specific-iterated BLAST. The interface to this web-based servece is in the module WEBBLAST2. The following functions are provided by this module.

- `webblast2-blastp-detail: (#name: string, #db: string, #level: num) -> string`

  Make connections to the blastp programiin NCBI website. The number of connections is given by the `#level` field. `#db` field indicates the database you want to BLAST against. These connections produce complete report of BLAST2 hits.

  For example, `webblast2-blastp-detail (#name: "myblast", #db: "nr", #level: 1)` makes a server connection myblast available. After you have done this, you can send your sequenct to the BLAST2 server like this:

  ```
  process "YOUR QUERY SEQUENCE" using myblast;
  ```

  The output of this has type `{(#uid: num, #accession: string, #title: string, #hits: [(#score: num, #pscore: num, #identities: num, #positives: num, #querystart: num, #queryend: num, #subjectstart: num, #subjectend: num, #alignment: string)])}`. The interpretation is similar to that of the WU-BLAST2 interface.

- `webblast2-blastn-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast2-blastp-detail`, but connect to blastn instead.

- `webblast2-blastx-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast2-blastp-detail`, but connect to blastx instead.

- `webblast2-tblastn-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast2-blastp-detail`, but connect to tblastn instead.

- `webblast2-tblastx-detail: (#name: string, #db: string, #level: num) -> string`

  Similar to `webblast2-blastp-detail`, but connect to tblastx instead.

B.36 NCBI COGs

This subsection is written by Chen Jing.

NCBI's Clusters of Orthologous Groups (COGs) were delineated by comparing protein sequences encoded in 7 complete genomes, representing 5 major phylogenetic lineages. Each COG consists of individual proteins or groups of paralogs from at least 3 lineages and thus corresponds to an ancient conserved domain. NCBI also provides a seb service for the comparison of user sequence to COG database. The URL is `http://www.ncbi.nlm.nih.gov/ COG/ cognitor.html`.

The Kleisli module WEBCOG is a driver to the server. The interface is given below.

- `webcog: (#seq: string) -> {(#accession: string, #title: string, #hits: [(#score: num, #expect: num, #identities: num, #positives: num, #querystart: num, #queryend: num, #subjectstart: num, #subjectend: num, #alignment: string)])}`

  The interpretation of the output type is similar to that of the WU-BLAST2 interface.

  Here is an example:

```
  webcog ( #seq: "YELLYHLALLERRQGDLDAAIDRYQAALQEPVDA");
  Result:
  {(#accession: "slr0626",
    #title: "2966735..2967931",
    #hits: [(#score: 29.0,
             #expect: 0.27,
             #identities: 46,
             #positives: 55,
             #querystart: 0,
             #queryend: 31,
             #subjectstart: 241,
             #subjectend: 272,
             #alignment: "YELLYHLALLERRQGDLDAAIDRYQAALQEPV
+EL YHLA   RRQ   + A   YQ AL E +
FELHYHLANAYRRQQKWELARKHYQKALDEEI
")]),
   ...}
```

B.37 NCBI UniGene

This subsection is written by Chen Jing.

NCBI UniGene is an experimental system for automatically partitioning GenBank sequences into a non-redundant set of gene-oriented clusters. Each UniGene cluster contains sequences that represent a unique gene, as well as related information such as the tissue types in which the gene has been expressed and map location. At present, only sequences from human and mouse have been processed in this system. The interface to this web-based service is in the module WEBUNIGENE. Here are the functions provided by the interface.

- `webunigene-summary: (#org: string, #keyword: string) -> {(#unigene: string, #title: string, #symbol: string)}`

  Search UniGene by a keyword. `#org` specifies the organism, it can be either "Hs" (Human) or "Mm" (mouse). The query terms you want to search is given in field `#keyword`. They can be ordinary words like "kinase" or specific identifiers, such as GenBank accession numbers. can also use `@chr(num)` as a search term to searh a particular chromosome or use `@lib(num)` to search a library. More tips can be found at `http://inhouse.ncbi.nlm.nih.gov/ cgi-bin/ UniGene/ query?ORG=Hs`.

- `webunigene-get-cluster: string ->{(#cid: string, #species: string, #title: string, #symbol: string, #description: {<#expression: (#source: { string}, #SAGEcid: string), #similarities: (#organism: string, #uid: num, #title: string, #identity: num, #length: num), #mapping: (#name: string, #value: string), #mrna: (#sid: num, #accn: string, #title: string, #protein-similarity: bool, #polyaden: bool, #sts: bool, #cgap: bool), #est: (#sid: num, #accn: string, #title: string, #source: string, #note: string, #clone-size: num, #protein-similar #polyaden: bool, #sts: bool, #cgap: bool)>})}`

  Given the unigene id, return the report of this cluster. For example:

  ```
  webunigene-get-cluster ( "Hs.210" );
  ```

  In the output, `#protein-similarity` is true when the sequence has similarity to known Proteins (after translation). `#polyaden` is true when a poly-Adenylation signal is contained in the sequence. `#sts` is true when a mapped Sequence-tagged site (STS) is contained. `#cgap` is true when Clone source is a CGAP library.

B.38 NCBI dbEST

This subsection is written by Chen Jing.

dbEST is a division of GenBank that contains sequence data and other information on ”single-pass” cDNA sequences, or Expressed Sequence Tags, from a number of organisms. The interface to these driver routines is in the Kleisli module WEBDBEST.

- `webdbest-get-summary-by-accn : string -> {(#id: num, #dbest_id: string, #lib_id: string, #lib_name: string)}`

  Given an accession number or a specification, return summaries that satisfy it. dbEST document id is given in field `#id`.

- `webdbest-doc-id-by-clone-id : string -> { num }`

  Given a clone id, return all related dbEST document ids.

- `webdbest-get-document-by-id : num ->{(#dbest_id: num, #est_name: string, #accession: string, #uid: num, #clone_id: string, #dna_type: string, #seq: string, #lib: (#id: num, #name: string, #organism: string, #sex: string, #organ: string, #tissue: string, #stage: string, #lab_host: string, #vector: string, #rsite1: string, #rsite2: string, #desc: string))}`

  Given a dbEST document id, return its document.

B.39 William R. Pearson’s SSEARCH3

This subsection is written by Wang Jiren.

Ssearch3 is the searching program for comparison a protein or DNA sequence to a sequence database using the Smith-Waterman algorithm developed by William R. Pearson in the University of Virginia.

The interface for William R. Pearson's SSEARCH3 driver is in the module SSEARCH3. By the way, this module also interfaces to FASTA3. It provides the functions listed below:

- `localfasta3(#name: string, #database: string, #numbestscore: num, #numbestalign: num, #evaluelimit: num, #dna: bool, #level: num) -> string`

  Similar to `localssearch3`, but connects to FASTA3 instead.

- `localssearch3(#name: string, #database: string, #numbestscore: num, #numbestalign: num, #evaluelimit: num, #dna: bool, #level: num) -> string`

  Establish connections to SSEARCH3. The number of connections is given by the `#level` field. The name of the connections is specified by the `#name` field. The field `#database` is the database name to be compared. The database is stored in the fasta format. The fields `#evaluelimit`, `#numbestscore,` and `#numbestalign` represent expectation value limit for displaying scores and alignments (typically 10.0 for protein sequence comparisons and 2.0 for DNA sequence comparisons), the number of best scores to show, and the number of best alignments to show, respectively. The field `#dna` is set to true if the comparison is for DNA sequences and to false if the comparison is for protein sequences. Please note if the value of field `#numbestscore`, `#numbestalign`, or `#evaluelimit` is set to equal or less than zero, then the default value will be used.

  Below is an example:

  ```
  localssearch3(
  #name: "ssearch3",
  #database: "/usr49/BIC/jrwang/ssearch-jrwang/globins630.fa",
  #numbestscore: 4, #numbestalign: 3,
  #evaluelimit: 0, #dna: false, #level: 1);
  ```

  After establishing the connection to SSEARCH3, you can do the search by sending request to this connection in the following method:

  ```
   process seq-string using ssearch3;
  ```

  Above seq-string is a DNA or protein sequence. The output has the type `{(#name: string, #z-score: num, #expect: num, #score: num, #identity: num, #overlap: num, #querystart: num, #queryend: num, #querydisplaystart: num, #subjectstart: num, #subjectend: num, #subjectdisplaystart: num, #subjectlength: num, #alignment: string)}`. As there are no fields `#z-score` and `#expect` for DNA comparison, these two fields are set to -999. The fields `#querystart`, `#queryend`, `#subjectstart`, and `#subjectend` are the starting and ending positions of query sequence and matched sequence in the database. The field `#subjectlength` is the length of this matched sequence in the database. These two fields `#querydisplaystart` and `#subjectdisplaystart` are the starting matched positions in the field `#alignment` for query and database sequence.

  The example of this request and output are listed below:

```
   process "MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVR" using ssearch3;
   Result:
{(#name: "BAHG_VITSP", #z-score: 325.6, #expect: 1.6E~13,
  #score: 253, #identity: 100.0, #overlap: 40, #querystart: 2,
  #queryend: 40, #querydisplaystart: 1, #subjectstart: 5,
  #subjectend: 41, #subjectdisplaystart: 1, #subjectlength: 152,
  #alignment: "MLDQQTINIIKATVPVLKEHG...
               :::::::::::::::::::::...
               MLDQQTINIIKATVPVLKEHG..."),
  ... }
```

Here is another example for DNA. The connection, request, and output for this example are:

```
localssearch3(
  #name: "localsearch3",
  #database: "/usr49/BIC/jrwang/ssearch-jrwang/orf_coding.fa",
  #numbestscore: 4, #numbestalign: 2,
  #evaluelimit: 0, #dna: true, #level: 1);
```

Request:

```
process "TGTCAATGACGTAATAGGAATTGAGCTGCTATCAGACTCTTTGTTA..."
using localsearch3;
Result:
 (#name: "YAL011W", #z-score: ~999, #expect: ~999,
  #score: 68, #identity: 64.865, #overlap: 37, #querystart: 22,
  #queryend: 58, #querydisplaystart: 1, #subjectstart: 333,
  #subjectend: 369, #subjectdisplaystart: 303, #subjectlength: 1917,
  #alignment: "TGTCAATGACGTAATAGGAATTGAGCTGCTATCAGACT...
                                    :: : :::: :  ::::...
      GCTTACTCATCCGCAATCAATTAAAAATTCTGGGGTGCTTTACGACT...")}
```

- `localfasta3-opt(#name: string, #database: string, #option: string, #level: num) -> string`

  Similar to `localssearch3-opt`, but connect to FASTA3 instead.

- `localssearch3-opt(#name: string, #database: string, #option: string, #level: num) -> string`

  The field `#option` let users specify some command line paremeters. For example, "-n" is used to force query to nucleotide sequence, and "-s BL62" is used to specify substitution matrix BLOSUM62. The meaning of fields `#name`, `#database`, and `#level` are the same as above `localssearch3`.

  Below is an example of this command:

```
  localssearch3-opt(
    #name: "ssearch3",
    #database: "/usr49/BIC/jrwang/ssearch-jrwang/globins630.fa",
    #option: "-s BL62", #level: 1);
```

and an example of its use:

```
  process "MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVR" using ssearch3;
  Result:
{(#name: "BAHG_VITSP", #z-score: 358.8, #expect: 2.2E~15,
  #score: 205, #identity: 100.0, #overlap: 40, #querystart: 1,
  #queryend: 40, #querydisplaystart: 1, #subjectstart: 1,
  #subjectend: 40, #subjectdisplaystart: 1, #subjectlength: 146,
  #alignment: "MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVR
               ::::::::::::::::::::::::::::::::::::::::
               MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVR...")}
```

B.40 Jiren's Codon Usage Program

Jiren wrote a program to tabulate codon usage. The interface to his program is the module CODONUSAGE. It provides the following function

- `codon-usage: [ string] -> {(#aa: string, #codons: {(#codon: string, #freq: num)})}`

  Each item in the input list is a data sequence. Each such sequence is assumed to be in frame. The codon usage table is then produced. For example,

```
codon-usage [
  "AAACGAATTTCAGTGATGAATTTGAATA...",
  "AAAGCTGCTGATGCAATGGAGGAATGGA...",
  ... ];

Result:
{(#aa: "A",
  #codons: {(#codon: "GCA", #freq: 0.222222222222222),
            (#codon: "GCC", #freq: 0.222222222222222),
            (#codon: "GCG", #freq: 0),
            (#codon: "GCT", #freq: 0.555555555555556)}),
 (#aa: "C",
  #codons: {(#codon: "TGC", #freq: 0),
            (#codon: "TGT", #freq: 0)}),
 (#aa: "D",
  #codons: {(#codon: "GAC", #freq: 0.4),
            (#codon: "GAT", #freq: 0.6)}),
... }
```

- `codon-usage2: [ string ] -> {(#type: string, #freqs: {(#name: string, #freq: num)})}`

  Produce frequencies of nucleotides and amino acids. For example,

  ```
  codon-usage2 [
    "AAACGAATTTCAGTGATGAATTTGAATA...",
    "AAAGCTGCTGATGCAATGGAGGAATGGA...",
    ... ];

  Result:
  {(#type: "na",
    #freqs: {(#name: "A", #freq: 0.37037037037037),
             (#name: "C", #freq: 0.0925925925925926),
             (#name: "G", #freq: 0.277777777777778),
             (#name: "T", #freq: 0.259259259259259)}),
   (#type: "aa",
    #freqs: {(#name: "A", #freq: 0.166666666666667),
             (#name: "C", #freq: 0),
             (#name: "D", #freq: 0.0555555555555556),
             ...
             (#name: "W", #freq: 0.0555555555555556),
             (#name: "Y", #freq: 0),
             (#name: "*", #freq: 0)})}
  ```

- `codon-usage3: [ string] -> (#total-dna: num, #dna-freq: {(#name: string, #count: num, #freq: num)}, #aa-freq: {(#name: string, #count: num, #freq: num)}, #codon-usage: {(#aa: string, #codons: {(#codon: string, #count: num, #freq: num)})})`

  Each item in the input list is a data sequence. Each such sequence is assumed to be in frame. The codon usage table, nucleotide counts and frequencies, and amino acid counts and frequencies are then produced. For example,

  ```
  codon-usage3 [
    "AAACGAATTTCAGTGATGAATTTGAATA...",
    "AAAGCTGCTGATGCAATGGAGGAATGGA...",
    ... ];

  Result:
  (#total-dna: 54,
   #dna-freq: {
     (#name: "A", #count: 20, #freq: 0.37037037037037),
     (#name: "C", #count: 5, #freq: 0.0925925925925926),
     ...},
   #aa-freq: {
     (#name: "A", #count: 3, #freq: 0.166666666666667),
     (#name: "C", #count: 0, #freq: 0),
     ...},
  ```

```
#codon-usage: {
   (#aa: "A",
    #codons: {
      (#codon: "GCA", #count: 1, #freq: 0.333333333333333),
      (#codon: "GCC", #count: 0, #freq: 0),
      (#codon: "GCG", #count: 0, #freq: 0),
      (#codon: "GCT", #count: 2, #freq: 0.666666666666667)}),
   ...})
```

- `codon-gen: (#code: {(#aa: string, #codon: string)}, #seq: string) -> string`

  Translate a protein sequence into a DNA sequence using the specified encoding.

- `codon-gen2: (#code: {(#aa: string, #codons: {(#codon: string, #freq: num)})}, #seq: string) -> string`

  Translate a protein sequence into DNA probabilistically using the specified codon usage table.

B.41 Washington Univ in St. Louis' HMM Search

This subsection was written by Wang Jiren.

Pfam is a large collection of multiple sequence alignments and hidden Markov models covering many common protein domains. Pfam HMM Search can be used for analyzing a query sequence to find Pfam domain matches. Web version Pfam HMM Search can be found at `http://pfam.wustl.edu/hmmsearch.shtml`.

The interface to web Pfam HMM search's data driver is in the module WEBHMMSEARCH. It provides the function listed below:

- `webhmmsearch: (#name: string, #fragment: bool) -> string`

  Establish connection to the HMM Search server at Washington University in St. Louis. The name of the connection is specified by the `#name` field. The `#fragment` field is used to determine whether current search is for standard Pfam search or for fragment Pfam search. The standard Pfam search is the most sensitive method for finding complete domains. The standard Pfam search may miss partial domains. If nothing is found with the standard search, the fragment Pfam search can be used, especially for the case that your sequence is a known fragment. Set `#fragment` field false for standard Pfam search and set `#fragment` field true for fragment Pfam search. Below is an example:

  ```
  webhmmsearch(#name: "webhmmsearch", #fragment: false);
  ```

  After establishing the connection, you can do the search by sending request to this connection in the following method:

  ```
  process sequence-string using webhmmsearch;
  ```

Above `sequence-string` is a protein sequence to be searched. The output has the type `{(#model: string, #start: num, #end: num, #score: num, #e-value: num, #alignment: string)}`. The `#model` field is the name of model the sequence belongs to. The `#start` and `#end` fields are the starting and ending positions of query sequence in the matched alignment and `#alignment` field is the result of matched alignment. The `#score` and `#e-value` fields are the values of score and e-value. An example of such a request and output are:

```
process "SSSTTLDCHANGVPEPQITWFKNNHKI...." using webhmmsearch;

Result:
{(#model: "ig",
  #start: 0,
  #end: 58,
  #score: 36.8,
  #e-value: 4.8E~7,
  #alignment: "GesvtLtCsvsgfgppgvsvtWyfk....
                +s tL C ++  g p +++tW+ k....
               SSSTTLDCHAN--GVPEPQITWF-K...")}
```

B.42 ExPASy's ENZYME

ENZYME is a repository of information relative to the nomenclature of enzymes. It is primarily based on the recommendations of the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology (IUBMB) and it describes each type of characterized enzyme for which an EC (Enzyme Commission) number has been provided. See `http://expasy.hcuge.ch/sprot/enzyme.html`. We have an interface to this website. The Kleisli module is WEBENZYME.

- `webenzyme-get-by-ec: string -> {(#ec_num: string, #official_name: [ string], #alias: [ string], #reaction: (#substrate: [ string], #product: [ string], #cofactor: [ string]), #comment: [string])}`

  Given an EC number, return the Enzyme information.

C Available External Primitives

In this appendix I list the external primitives that have been registered for use CPL. I group them by the Kleisli modules that implement them. See *The Kleisli Query System Reference Manual* for more information.

C.1 General Operations

These are implemented in the Kleisli module GEN.

- `fix : (('1 -> '2) -> '1 -> '2) -> '1 -> '2`

  This is the general fixpoint operator. That is, it satisfies the axiom $\texttt{fix}(f) = f(\texttt{fix}(f))$. For example, the standard factorial function can be defined in terms of `fix` as follows:

  ```
  primitive factorial == fix (\f => \n => if n < 1 then 1 else n * f (n - 1));
  ```

- `error: string -> '1`

  Raise a runtime error. The string is used as the error message.

- `handle: (unit -> '1, string -> '1) -> '1`

  This is a error handling function. $\texttt{handle}(F, H)$ works like this. First, $F()$ is executed. If some $\texttt{error}(S)$ is raised during the execution, then $H(S)$ is executed. The final result is either the value returned by $F()$ if there is no error, or is the value returned by $H(S)$ if $\texttt{error}(S)$ is raised.

- `before: ('1,  '1 -> '2) -> '2`

  This is a sequencing operator. $\texttt{before}(F, H)$ has the same semantics as the function application as $F.H$ if both F and H have no side effect. It differs from ordinary function application by ensuring that F is evaluated before being passed to H.

- `parallel: (unit -> '2, unit -> '1) -> ('2, '1)`

  This is a parallel operator. $\texttt{parallel}(F, H)$ has the same semantics as forming the pair $(F(), H())$. It differs from ordinary pair formation by ensuring that $F()$ and $H()$ are executed in parallel.

C.2 Set Operations

These are implemented in the Kleisli module SET.

- `set-trim: { ''1 } -> { ''1 }`

  Semantically, this is the identity function on sets. However, it has the side effect of forcing the set to be evaluated until at least one element is known.

- `set-unique: { ''1 } -> { ''1 }`

  Semantically, this is the identity function on sets. However, it has the side effect of removing duplicates from the representation.

- `set-subsetn: ({ ''1 }, num) -> { { ''1 } }`

  Generate all subsets having the given number of elements.

- `set-powersetn: ({ ''1 }, num) -> { { ''1 } }`

  Generate all subsets having at most the given number of elements.

- `set-limitedpowerset: ({ ''1 }, num) -> ({ { ''1 } }, { { ''1 } })`

  Semantically, `set-limitedpowerset`$(X, n) = ($ `set-powersetn`(X, n), `set-subsetn`$(X, n))$. However, it is more efficient.

- `set-sri: ((''2, '1) -> '1, '1) -> { ''2 } -> '1`

  Structural recursion on the insert presentation of sets. Hence `set-sri @` (i, e) `@` $\{o_1, \ldots, o_n\} = o_1\ i\ \ldots\ o_n\ i\ e$. Note that this is valid only if i is commutative and idempotent.

- `set-sru: (('1, '1) -> '1, ''2 -> '1, '1) -> { ''2 } -> '1`

  Structural recursion on the union presentation of sets. Hence `set-sru @` (u, f, e) `@` $\{o_1, \ldots, o_n\} = f(o_1)\ u\ \ldots\ u\ f(o_n)\ u\ e$. Note that this is valid only if u is commutative, idempotent, and associative.

- `set-hom: (('1, '1) -> '1, ''2 -> '1, '1) -> { ''2 } -> '1`

  Similar to `set-sru`, but clean up the set first. Hence `set-hom @` (u, f, e) `@` $\{o_1, \ldots, o_n\} = f(o_1)\ u\ \ldots\ u\ f(o_n)\ u\ e$. Note that this is valid only if u is commutative and associative. That is, u needs not be idempotent.

- `set-isempty: { ''1 } -> bool`

  Check if the set is empty.

- `set-head: { ''1 } -> ''1`

  Extract the first element from the set. (Positions in a set is defined at runtime and objects are allowed to repeat.)

- `set-tail: { ''1 } -> { ''1 }`

  Remove the first element from the set.

- `set-firstn: (num, { ''1 }) -> { ''1 }`

  Extract the first n elements of the set.

- `set-thereis: (''1 -> bool) -> { ''1 } -> bool`

  Is there an element in the set that satisfies the predicate?

- `set-forall: (''1 -> bool) -> { ''1 } -> bool`

  Does every element in the set satisfy the predicate?

- `set-numitems: { ''1 } -> num`

  Return the cardinality of the set.

- `set-roughcount: { ''1 } -> num`

  Estimate an upperbound on the cardinality of the set. More efficient than `set-numitems`.

- `set-member: (''1, { ''1 }) -> bool`

  Set membership test.

- `set-subset: ({ ''1 }, { ''1 }) -> bool`

  Subset test.

- `set-intersect: ({ ''1 }, { ''1 }) -> { ''1 }`

  Set intersection.

- `set-diff:  ({ ''1 }, { ''1 }) -> { ''1 }`

  Set difference.

- `set-index: (''2 -> ''1,  { ''2 },  (#lt: ''1 -> { ''2 },  #leq: ''1 -> { ''2 }, #eq: ''1 -> { ''2 }, #geq: ''1 -> { ''2 }, #gt: ''1 -> { ''2 }) -> '3) -> '3`

  Query a set using an index. `set-index`(I, S, P) works like this. First an index is created on S using I as the indexing function. Let a, b, c, d, e the associated access functions to this index that return objects that are "less than," "less than or equal to," "equal to," "greater than or equal to," and "greater than" the keys. Then apply P to these access functions. For example, here is how we can define an indexed join on the name field of two relations.

  ```
  primitive probe == \R => (#eq: \p, ...) => {(x, y) | \x <- R, \y <- x.#name.p};
  primitive index-join == (\S, \R) => set-index (#name, S, R.probe);
  ```

- `set-index': (''2 -> ''1, { ''2 }) -> (#lt: ''1 -> { ''2 }, #leq: ''1 -> { ''2 }, #eq: ''1 -> { ''2 }, #geq: ''1 -> { ''2 }, #gt: ''1 -> { ''2 })`

  Create an in-memory index on a set. `set-index`(I, S, P) is equivalent to $P($`set-index'`$(I, S))$.

C.3 Bag Operations

These are implemented in the Kleisli module BAG.

- `bag-trim: {| ''1 |} -> {| ''1 |}`

  Semantically, this is the identity function on bags. However, it has the side effect of forcing the bag to be evaluated until at least one element is known.

- `bag-insertn: (num, ''1, {| ''1 |}) -> {| ''1 |}`

  Insert n copies of an object into a bag.

- `bag-sri: ((''2, '1) -> '1, '1) -> {| ''2 |} -> '1`

  Structural recursion on the insert presentation of bags. Hence `bag-sri @` (i, e) `@` $\{|o_1, \ldots, o_n|\} = o_1\ i\ \ldots\ o_n\ i\ e$. Note that this is valid only if i is commutative.

- `bag-sru: (('1, '1) -> '1, ''2 -> '1, '1) -> {| ''2 |} -> '1`

  Structural recursion on the union presentation of bags. Hence `bag-sru @` (u, f, e) `@` $\{|o_1, \ldots, o_n|\} = f(o_1)\ u\ \ldots\ u\ f(o_n)\ u\ e$. Note that this is valid only if u is commutative and associative. For example, to sum up a bag of numbers,

  ```
  primitive total == bag-sru @ (+, \x => x, 0);
  ```

- `bag-hom:  (('1, '1) -> '1, ''2 -> '1, '1) -> {| ''2 |} -> '1`

  Same as `bag-sru`, but eliminate duplicates from the bag first.

- `bag-isempty: {| ''1 |} -> bool`

  Is the bag empty?

- `bag-head: {| ''1 |} -> ''1`

  Extract the first element from the bag. (Positions in a bag is defined at runtime.)

- `bag-tail: {| ''1 |} -> {| ''1 |}`

  Remove the first element from the bag.

- `bag-firstn: (#1: num, #2: {| ''1 |}) -> {| ''1 |}`

  Extract the first n elements from the bag.

- `bag-thereis: (''1 -> bool) -> {| ''1 |} -> bool`

  Is there an element of the bag satisfying the predicate?

- `bag-forall: (''1 -> bool) -> {| ''1 |} -> bool`

  Does every element of the bag satisfy the predicate?

- `bag-numitems: {| ''1 |} -> num`

  Number of items in the bag. For example, to find the average of a bag of numbers,

  ```
  primitive average == \b => b.total / b.bag-numitems;
  ```

- `bag-member: (''1, {| ''1 |}) -> bool`

  Bag membership test.

- `bag-subbag: ({| ''1 |}, {| ''1 |}) -> bool`

  Subbag test.

- `bag-min: ({| ''1 |}, {| ''1 |}) -> {| ''1 |}`

  Bag "intersection" in a minimal way. The number of times an object occurs in the intersection is equal to the minimum of its occurrences in the two input bags.

- `bag-max: ({| ''1 |}, {| ''1 |}) -> {| ''1 |}`

  Bag "intersection" in a maximal way. The number of times an object occurs in the intersection is equal to the maximum of its occurrences in the two input bags.

- `bag-monus: ({| ''1 |}, {| ''1 |}) -> {| ''1 |}`

  Bag subtraction.

- `bag-unique: {| ''1 |} -> {| ''1 |}`

  Eliminate duplicates from the bag.

- `bag-index: (''2 -> ''1,{| ''2 |}, (#lt: ''1 -> {| ''2 |},#leq: ''1 -> {| ''2 |}, #eq: ''1 -> {| ''2 |},#geq: ''1 -> {| ''2 |},#gt: ''1 -> {| ''2 |}) -> '3) -> '3`

  Query a bag using an index. `bag-index`(I, S, P) works like this. First an index is created on S using I as the indexing function. Let a, b, c, d, e the associated access functions to this index that return objects that are "less than," "less than or equal to," "equal to," "greater than or equal to," and "greater than" the keys. Then apply P to these access functions. For example, here is how we can define an indexed join on the name field of two bags.

```
primitive probe == \R => (#eq: \p,...) =>
   {|(x, y) | \x <-- R, \y <-- x.#name.p|};
primitive index-join == (\S, \R) => bag-index (#name, S, R.probe);
```

- `bag-index': (''2 -> ''1, {|''2|}) -> (#lt: ''1 -> {|''2|}, #leq: ''1 -> {|''2|}, #eq: ''1 -> {| ''2 |}, #geq: ''1 -> {| ''2 |}, #gt: ''1 -> {| ''2 |})`

  Create an in-memory index on a bag. `bag-index`(I, S, P) is equivalent to $P($`bag-index'`$(I, S))$.

C.4 List Operations

These are implemented in the Kleisli module LIST.

- `list-trim: [ ''1 ] -> [ ''1 ]`

  This is semantically the identity function on lists. But it has the side effect of evaluating a list until its head is known.

- `cons: (''1, [ ''1 ]) -> [ ''1 ]`

  Insert an item to the front of the list.

- `snoc:  (''1, [ ''1 ]) -> [ ''1 ]`

  Insert an item to the back of the list.

- `list-sri: ((''2, '1) -> '1, '1) -> [ ''2 ] -> '1`

  Structural recursion on the insert presentation of lists. Hence `list-sri @` (i, e) `@` $[o_1, \ldots, o_n] = o_1\ i\ \ldots\ o_n\ i\ e$.

- `list-irs: ((''2, '1) -> '1, '1) -> [ ''2 ] -> '1`

  Same as `list-sri`, but reverse the list first.

- `list-sru: (('1, '1) -> '1, ''2 -> '1, '1) -> [ ''2 ] -> '1`

  Structural recursion on the union presentation of lists. Hence `list-sru @` (u, f, e) `@` $[o_1, \ldots, o_n]$ $= f(o_1)\ u\ \ldots\ u\ f(o_n)\ u\ e$. Note that this is valid only if u is associative.

- `list-isempty: [ ''1 ] -> bool`

  Is the list empty?

- `list-head: [ ''1 ] -> ''1`

  Extract the first element of the list.

- `list-tail: [ ''1 ] -> [ ''1 ]`

  Remove the first element of the list.

- `list-firstn: (num, [ ''1 ]) -> [ ''1 ]`

  Extract the first n element of the list.

- `list-thereis: (''1 -> bool) -> [ ''1 ] -> bool`

  Does the list has an element that satisfies the predicate?

- `list-forall: ''1 -> bool) -> [ ''1 ] -> bool`

  Does every element of the list satisfy the predicate?

- `list-numitems: [ ''1 ] -> num`

  The number of items on the list.

- `list-member: (''1, [ ''1 ]) -> bool`

  List membership test.

- `list-uniquesortdesc: [ ''1 ] -> [ ''1 ]`

  Sort the list into descending order, as well as remove duplicates.

- `list-sortdesc: [ ''1 ] -> [ ''1 ]`

  Sort the list into descending order, but retain duplicates.

- `list-uniquesort: [ ''1 ] -> [ ''1 ]`

  Sort the list into ascending order, as well as remove duplicates.

- `list-sort:  [ ''1 ] -> [ ''1 ]`

  Sort the list into ascending order, but retain duplicates.

- `list-rev: [ ''1 ] -> [ ''1 ]`

  Reverse the list.

- `list-gensort: ((''1, ''1) -> bool) -> [ ''1 ] -> [ ''1 ]`

  Sort the list into ascending ordering using the given comparison function. For example

  ```
  list-gensort @ ((\x, \y) => x > y) @ [1,2,3,4,3,2,1];
  ```

  produces the sorted list `[1, 1, 2, 2, 3, 3, 4]`.

- `list-genuniquesort: ((''1, ''1) -> bool) -> [ ''1 ] -> [ ''1 ]`

  Same as `list-gensort`, but also eliminates duplicates.

- `list-prefix: (num, [ ''1 ]) -> [ ''1 ]`

  Extract the first n elements of the list.

- `list-suffix: (num, [ ''1 ]) -> [ ''1 ]`

  Skip the first n elements of the list and return the rest.

- `list-postfix: (num, [ ''1 ]) -> [ ''1 ]`

  Extract the last n items of the list.

- `list-sublist: (num, num, [ ''1 ]) -> [ ''1 ]`

  Extract a contiguous segment of the list. So `list-sublist`(n, m, l) skips the first n items in l and return the next m items.

- `list-index: (''2 -> ''1, [ ''2 ], (#lt: ''1 -> [ ''2 ], #leq: ''1 -> [ ''2 ], #eq: ''1 -> [ ''2 ], #geq: ''1 -> [ ''2 ], #gt: ''1 -> [ ''2 ]) -> '3) -> '3`

  Query a list using an index. `list-index`(I, S, P) works like this. First an index is created on S using I as the indexing function. Let a, b, c, d, e the associated access functions to this index that return objects that are "less than," "less than or equal to," "equal to," "greater than or equal to," and "greater than" the keys. Then apply P to these access functions. For example, here is how we can define an indexed join on the name field of two lists.

  ```
  primitive probe == \R => (#eq: \p,...) =>
     [(x, y) | \x <--- R, \y <--- x.#name.p];
  primitive index-join == (\S, \R) => list-index (#name, S, R.probe);
  ```

- `list-index': (''2 -> ''1, [ ''2 ]) -> (#lt: ''1 -> [ ''2 ], #leq: ''1 -> [ ''2 ], #eq: ''1 -> [ ''2 ], #geq: ''1 -> [ ''2 ], #gt: ''1 -> [ ''2 ])`

  Create an in-memory index on a list. `list-index`(I, S, P) is equivalent to $P($`set-index'`$(I, S))$.

C.5 Numeric Operations

These are implemented in the Kleisli module NUM.

- `log: (num, num) -> num`

  `log`$(x, y) = log_x\ y$.

- `logE: num -> num`

  `logE`$(x) = log_e\ x$.

- `exp: num -> num`

  `exp`$(x, y) = x^y$.

- `expE: num -> num`

  `expE`$(x) = e^x$.

- `+: (num, num) -> num`

  Addition of numbers.

- `-: (num, num) -> num`

  Subtraction of numbers.

- `*: (num, num) -> num`

  Multiplication of numbers.

- `mod: (num, num) -> num`

  Modulus of numbers. For example, `5 mod 3` is `2`.

- `div: (num, num) -> num`

  Integer division of numbers. For example, `5 div 3` is `1`.

- `/: (num, num) -> num`

  Real division of numbers. For example, `5 / 3` is `1.66666666666667`.

- `~: num -> num`

  Negate a number.

- `num-stringify: num -> string`

  Convert a number into a string. For example, `num-stringify 123.444` is `"123.444"`.

- `num-realfromstring: string -> num`

  Convert a string into a number. The string is expected to be a real number. For example, `num-realfromstring "123.33"` is `123.33`.

- `num-intfromstring: string -> num`

  Convert a string into a number. The string is expected to be an integer. For example, `num-intfromstring "123"` is `123`.

C.6 String Operations

These are implemented in the Kleisli module STRING.

- `string-fmt: (#fmt: string, #arg: [<#int: num, #real: num, #bool: bool, #str: string, #int-l: (#1: num, #2: num), #real-l: (#1: num, #2: num), #bool-l: (#1: num, #2: bool), #str-l: (#1: num, #2: string), #int-r: (#1: num, #2: num), #real-r: (#1: num, #2: num), #bool-r: (#1: num, #2: bool), #str-r: (#1: num, #2: string)>]) -> string`

  This one let you do formatted printing the way C, C++, Perl do. It is best illustrated by some examples:

  ```
  string-fmt (
    #fmt: "%8.2f ** %4d ** %20s ** %b",
    #arg: [<#real: 123.456789>,
           <#int: 123.456>,
           <#str: "hello world">,
           <#bool: true>]);
  Result:
  "  123.46 **  123 **          hello world ** true"


  string-fmt (
    #fmt: "%8.2f ** %4d ** %s ** %b",
    #arg: [<#real-r: (10, 123.456789)>,
           <#int: 123.456>,
           <#str-l: (20, "hello world")>,
           <#bool: true>]);
  Result:
  "    123.46 **  123 ** hello world          ** true"
  ```

- `string-rev : string -> string`

  Reverse a string.

- `^: (string, string) -> string`

  String concatenation.

- `string-iswhite: string -> bool`

  Is the character a white character? That is, is it a space, a tab, or a newline?

- `string-isprefix: (string, string, num) -> bool`

  Prefix test. `string-isprefix`(s_1, s_2, n) works like this. Skip the first n characters of s_2. Then check if the remaining portion of s_2 starts with s_1. So `string-isprefix ("2,3", "12,3334", 1)` is true, but `string-isprefix ("2,3", "12,3334", 2)` is false.

- `string-suffix: (string, num) -> string`

  Skip the first n characters of the string and return the rest.

- `string-length: string -> num`

  The length of the string.

- `string-span: (string, num, num) -> string`

  Extract the substring spanning the specified positions. For example, `string-span("abc", 1, 2)` returns `"bc"`.

- `string-substring: (string, num, num) -> string`

  Extract a contiguous segment of the string. `string-substring`(s, n, m) skips the first n characters of s and returns the next m characters.

- `string-issubstring: (string, num, string) -> bool`

  Substring test. `string-issubstring`(s_1, n, s_2) skips the first n characters of s_1 then checks if s_2 occurs in the remaining portion of s_1. For example, `string-issubstring("12334", 1, "33")` is true, but `string-issubstring("12334", 3, "33")` is false.

- `string-islike: (string, string) -> bool`

  More sophisticated substring test. `string-islike`(s_1, s_2) allows s_2 to be a pattern of the form "%p_1%...%p_2%". The test is true iff the strings p_1, ..., p_n occur in s_1 in that order. If the leading % is omitted, then s_1 is required to start with p_1. If the trailing % is omitted, then s_1 is required to end with p_n.

- `string-trans: (string, string, string) -> string`

  String translation. `string-trans`(n, m, s) builds a map from characters in n to the corresponding characters in m and then applies this map to s. So `string-trans ("123", "000", "123456")` produces `"000456"`.

- `string-tokenize: (string, string, num) -> [ string ]`

  Break up the string using the specified separators after skipping the specified number of characters. Hence, `string-tokenize(",.", "i love you, you love me.", 0)` produces `["i love you", " you love me"]`.

- `string-chr: num -> string`

  Given ASCII code, produce the corresponding character.

- `string-ord: string -> num`

  Given a character, produce its ASCII code.

- `string-implode: [ string ] -> string`

  Concatenate the list of strings into a single string. So `string-implode ["h", "e", "l", "p"]` produces `"help`.

- `string-explode: string -> [ string ]`

  Break up the string into a list of characters. For example, `string-explode "help"` produces `["h", "e", "l", "p"]`.

- `string-tolower: string -> string`

  Convert to lowercase.

- `string-toupper: string -> string`

  Convert to uppercase.

- `string-newline: string`

  The newline character.

- `string-tab: string`

  The tab character.

- `string-quote: string`

  The string quotation character.

- `string-stdcplfromco: ''1 -> string`

  Convert a complex object into a string conforming to the Standard Kleisli Exchange Format, defined in *The Kleisli Query System Reference Manual*.

- `string-compressedcplfromco: ''1 -> string`

  Convert a complex object into a string conforming to the Compressed Kleisli Exchange Format, defined in *The Kleisli Query System Reference Manual*.

- `string-htmlfromco: ''1 -> string`

  Convert a complex object into a HTML formatted string.

- `string-stdcpltoco: string -> ''1`

  This is the inverse of `string-stdcplfromco` and `string-compressedcplfromco`.

C.7 Boolean Operations

These are implemented in the Kleisli module `BOOL`. They have the usual meanings.

- `andalso: (bool, bool) -> bool`
- `orelse: (bool, bool) -> bool`
- `not: bool -> bool`
- `butnot: (bool, bool) -> bool`

C.8 Operations on the New Type *array*

These are to be implemented in the proposed Kleisli module ARRAY to support a new type `array`. This is mainly to support multidimensional homogeneous arrays. These arrays are zero-based. A multidimensional array is implemented by storing its elements in a one-dimensional vector.

- `array-mk : (^2>=1, [''1]) -> array(^2>=1, ''1)`

  Create a k-dimensional array. The first component of the input indicates the size of each dimension. The second component is a list of objects used to populate the array in row-major order. The length of this list must equal the product of the dimension sizes. For example, `array-mk((2,2),[1,2,3,4])` creates the following 2-by-2 array:

1	2
3	4

- `array-km : array(^2>=1, ''1) -> (^2>=1, [''1])`

  The inverse of `array-mk`.

- `array-dim : array(^2>=1, ''1) -> ^2>=1`

  Extract the sizes of the dimensions of the array.

- `array-dimk : (array(^2>=1, ''1), num) -> num`

  Extract the size of the specified dimension of the array.

- `array-size : array(^2>=1, ''1) -> num`

  The number of elements in the array.

- `array-checkbound : (array(^2>=1, ''1), ^2>=1) -> bool`

  Is the index within the bounds of the array?

- `array-jump : (array(^2>=1, ''1), num) -> ''1`

  View the array as a one-dimensional vector and return the nth element of the vector.

- `array-offset : (array(^2>=1, ''1), ^2>=1) -> num`

  Given an index into the k-dimensional array, return equivalent index into the equivalent one-dimensional vector.

- `array-sub : (array(^2>=1, ''1), ^2>=1) -> ''1`

  Indexed into a k-dimensional array. It satisfies the axiom: `array-sub`(A, I) = `array-jump`$(A$, `array-offset`$(A, I))$. For example, `array-sub(array-mk((2,2),[1,2,3,4]),(0,1))` returns 2.

- `array-tabulate : (^2>=1, ^2>=1 -> ''1) -> array(^2>=1, ''1)`

  Create a k-dimensional array of the given size and populate it using the given function. For example,

```
primitive AnArray == array-tabulate((2,3), (\x, \y) => x + y);
```

defines `AnArray` to be the following 2-by-3 array:

0	1	2
1	2	3

- `array-subslab : (array(^2>=1,''1),^2>=1,^2>=1) -> array(^2>=1,''1)`

  Extract a slab of an array. The first input is the array. The second input is the pivot point. The last input specifies the size of the slab. For example, we can extract the last column in the above array by `array-subslab(AnArray, (0, 2), (2, 1))`.

- `array-graph : array(^2>=1, ''1) -> {(^2>=1, ''1)}`

  Generate the graph of an array. For example, `array-graph(AnArray)` is the set $\{((0,0),0)$, $((0,1),1)$, $((0,2),2)$, $((1,0),1)$, $((1,1),2)$, $((1,2),3)\}$.

- `array-sri : (^3>=1, (''2, ''1) -> ''1, ''1) -> {(^3>=1, ''2) } -> array(^3>=1, ''1)`

  Create an array from a partial "graph". It works like this: `array-sri`(d, i, e) takes in a graph G. It first creates an array A with dimensions specified by d. It initializes all elements of A to e. For for each (d', x) in G, it fetches the current d'th element y of A, and replaces it by $i(x, y)$. Note that the i above should be idempotent and commutative.

  For example, the Minkowski sum [14] that is frequently used in two-dimensional black-and-white image processing as a dilation operator can be defined as follow:

  ```
  primitive minkowski+ == (\A, \B) =>
    let ((\xa, \ya), (\xb, \yb)) == (array-dim (A), array-dim (B))
    in let \graph == {((ua + ub, va + vb), true) |
                           ((\ua, \va), true) <- array-graph (A),
                           ((\ub, \vb), true) <- array-graph (B)}
       in array-sri @ ((xa + xb, ya + yb), orelse, false) @ graph;
  ```

- `array-index+ : (^1>=1, ^1>=1) -> ^1>=1`

  Componentwise addition of two array indices.

- `array-index- : (^1>=1, ^1>=1) -> ^1>=1`

  Componentwise subtraction of the second index from the first index.

- `array-indexcheck : (^1>=1, ^1>=1) -> bool`

  True iff each component of the first index is less than that of the corresponding component of the second index. Satisfies the axiom `array-checkbound`$(A, I) =$ `array-indexcheck`(`array-dim`$(A), I)$.

- `array-indexoffset : (^1>=1, ^1>=1) -> num`

  Treat the first input as the dimensions of an array. Treat the second input as an index into that array. Return the equivalent index position when that array is viewed as a one-dimensional vector.

C.9 Operations on the New Type *het*

These are implemented in the Kleisli module HET to support a new type `het`. This is mainly to support wildcard search on a large complex structure.

- **`het-mk : ''1 -> het`**

  Inject a complex object into the heterogeneous type `het`.

- **`het-isnum: het -> bool`**

  Is the heterogeneous object numeric?

- **`het-isbool: het -> bool`**

  Is the heterogeneous object Boolean?

- **`het-isstring: het -> bool`**

  Is the heterogeneous object a string?

- **`het-isunit: het -> bool`**

  Is the heterogeneous object the unique object of type `unit`?

- **`het-isset: het -> bool`**

  Is the heterogeneous object a set?

- **`het-isbag: het -> bool`**

  Is the heterogeneous object a bag?

- **`het-islist: het -> bool`**

  Is the heterogeneous object a list?

- **`het-isfield: string -> het -> bool`**

  Is the heterogeneous object a record having the specified field?

- **`het-iscase:  string -> het -> bool`**

  Is the heterogeneous object a variant having the specified tag?

- **`het-isuser:  string -> het -> bool`**

  Is the heterogeneous object having the specified user-defined type?

- **`het-km: het -> ''1`**

  The inverse of `het-mk`.

- **`het-match: (string, het) -> { het }`**

  Extract the subcomponents of the heterogeneous object according to the specifier string. The specified string is interpreted as follows:

Specifier S	Heterogeneous Object H	`het-match`(S, H)
`?`	Set, bag, or list	Elements of H
`?%`	Variant with any tag	The value of H
`##`l	Record	The `#`l field of H
`%#`l	Variant with tag `#`l	The value of H
`*##l`	Anything	All `#`l fields of records anywhere in H
`*%#l`	Anything	All values of variants with tag `#`l anywhere in H

- **`het-multimatch: (string, het) -> { het }`**

  Extract the subcomponents of the heterogeneous object according to the extended specifier string. The extended specified string is a sequence of specifier strings separated by a dot. The semantics is $\texttt{het-multimatch("}s_1.\cdots.s_n\texttt{", } H) = \{y \mid \backslash x \texttt{ <- het-match("}s_1\texttt{", } H\texttt{)}, \backslash y \texttt{ <- het-multimatch("}s_2.\cdots.s_n\texttt{", } x\texttt{)}\}$.

D Available System Operations

In this appendix I list all external primitives that affect the configuration of the system. These primitives are highly unsafe. So make sure you know what you are doing when using them.

D.1 Operations to Look Up System Information

These are implemented in the Kleisli module INFO.

- `info-listallprim: '1 -> [ string ]`
  List all external primitives.
- `info-gettypeprim: string -> string`
  Return the type of the external primitive.
- `info-listallservers: '1 -> [ string ]`
  List all data drivers/servers.
- `info-getserverspec: string -> string`
  Return the request type of the server.

D.2 Operations to Run Scripts

These are implemented in the Kleisli module SCRIPTS.

- `usescript: string -> unit`
  Execute the CPL script in the specified file.
- `usescript': string -> unit`
  Execute the CPL script in the specified file, but only if it was never executed before.
- `usesmlscript: string -> unit`
  Execute the SML script in the specified file.

D.3 Operations to Do System Reconfiguration

These are implemented in the Kleisli module SETTINGS.

- `settings-traperr: bool -> bool`
  Should errors be automatically trapped in comprehension? Return the previous setting. Default is false.

- `settings-prompt: string -> string`

  What string is to be used as prompt? Return the previous prompt string.

- `settings-raisegoofedevenwhennotfatal: bool -> bool`

  Should all errors be made fatal? Return the previous setting. Default is false.

- `settings-printruntimeinfo: bool -> bool`

  Should runtime information be printed? Return the previous setting. Default is false.

- `settings-useruntimecmporder: bool -> bool`

  Should runtime ordering be used for comparison? Return the previous setting. Default is true.

D.4 Unsafe Operations

These are implemented in the Kleisli module UNSAFE.

- `delete-primitive: string -> bool`

  Delete the primitive or macro. Report the status of the deletion.

- `delete-server: string -> bool`

  Delete the server. Report the status of the deletion.

- `access-primitive: string -> '1`

  Treat the string as the name of a registered primitive. Return that primitive. For example, `(access-primitive "+") @ (4,5)` returns `9`.

- `access-server: string -> num -> '1 -> '2`

  Treat the string as the name of a registered server. Return that server function. For example, `(access-server (stdout)) @ 2 @ ("/tmp/sss", "num", 1)` creates the file `"/tmp/sss"` of type `num` and writes the number `1` to it. Here `2` is the priority of the server call.

- `force: ''1 -> ''1`

  It is semantically the identity function on complex object. But it has the side-effect of removing all laziness from its input. That is, it makes sure that its input becomes fully evaluated. Can be used with the `before` primitive to guarantee strict sequentiality.

- `materialize: string -> bool`

  Treat the string as the name of a primitive or macro. Cause it to be materialized. That is, cause it to be compiled immediately once and for all. Report the status of the materialization.

- `s2b: { ''1 } -> {| ''1 |}`

  Converts set to bag as efficiently as possible. It is not safe in the sense that it is nondeterministic. Basically, it makes sure the resulting bag has every element in the input set, but it does not ensure that each element appears exactly once in the resulting bag. That is, if $S_1 = S_2$, there is no guarantee that $\texttt{s2b}(S_1) = \texttt{s2b}(S_2)$.

- `b2s: {| ''1 |} -> { ''1 }`

  Converts bag to set as efficiently as possible. This one is actually deterministic and is safe. We guarantee that `b2s` is the inverse of `s2b`.

- `s2l: { ''1 } -> [ ''1 ]`

  Converts set to list as efficiently as possible. It is not safe because it is nondeterministic. It makes sure the resulting bag has every element in the input set, but it does not ensure that each element appears exactly once or appear in any fixed order in the bag.

- `l2s: [ ''1 ] -> { ''1 }`

  Inverse of `s2l`. It is a safe operation for converting list to set as efficiently as possible.

- `b2l: {| ''1 |} -> [ ''1 ]`

  Converts bag to list. Nondeterministic. It makes sure that every element of the input bag appears in the output list the same number of times it appears in the bag. But the order of appearance in not fixed.

- `l2b: [ ''1 ] -> {| ''1 |}`

  Inverse of `b2l`. Safe. Converts list to bag as efficiently as possible.