THE COLLECTION PROGRAMMING LANGUAGE
Reference Manual

Limsoon Wong

Institute of Systems Science
Heng Mui Keng Terrace, Singapore 119597
email: limsoon@iss.nus.sg

11 May 1996

Version C Release 16 April 1996

Contents

List of Figures

1 Introduction

1.1 Introduction

Based on some of the ideas described in [3], 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 [13] 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 [17] 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 [24]. 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 [15], but not in languages such as ML [13]. 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 [16], Remy [18], 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 [26, 2].

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 [3]. An alternative and less formal introduction to a variant of CPL can be found in [1]. 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.

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

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

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

CPL allows $(u_1, \cdots, u_n)$ as a syntactic sugar for $(\#1 : u_1, \cdots, \#n : u_n)$. 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 $(l_1 : u_1, \cdots, l_n : u_n)$ 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 [7] or Hull and Yap [9] for more information.) Objects of type $u \texttt{ -> } v$ 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 $u \texttt{ -> } v$ 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 [3]. 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 [20] 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 [20] 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 [20] 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 $\langle l_1 : \sigma_1, \cdots, l_n : \sigma_n;$ `''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.

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 $\langle l_1 : \beta_1, \cdots, l_n : \beta_n;$ `'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.

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

$$\frac{\begin{array}{c} l^{(l:u,\ l_1:u_1, \cdots, l_n:u_n)} : (l : u,\ l_1 : u_1, \cdots, l_n : u_n) \texttt{ -> } u \\ e_1 : (l : u,\ l_1 : u_1, \cdots, l_n : u_n) \qquad e_2 : u \end{array}}{\texttt{modify } l \texttt{ of } e_1 \texttt{ to } e_2 : (l : u,\ l_1 : u_1, \cdots, l_n : u_n)}$$

Figure 2: The constructs for record types in CPL.

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.

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

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

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

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.

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.

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

Figure 5: The constructs for set types in CPL.

For bag types, the expression constructs are given in Figure 6. The meaning of $\{||\}^s$ is the empty bag. The meaning of $\{|e|\}$ is the singleton bag containing e. The meaning of e_1 $\{|+|\}$ 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 $\{|+|\}$ e_2 is a bag of six apples and five oranges. The expression ext$\{|e_1 \mid \backslash x^t$ <-- $e_2|\}$ stands for the bag $e_1[o_1/x^t]$ $\{|+|\}$ $\cdots$ $\{|+|\}$ $e_1[o_n/x^t]$. More information on bags can be found in [11].

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

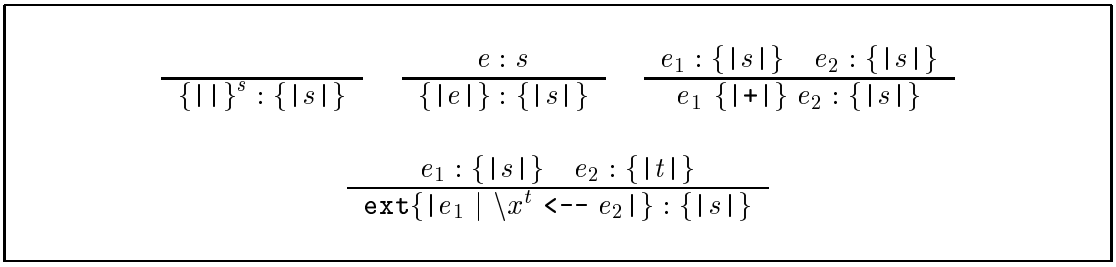

Figure 6: The constructs for bag types in CPL.

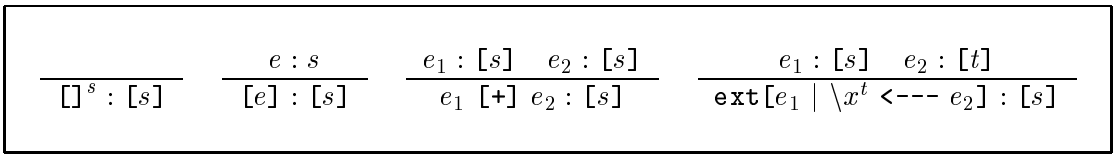

Figure 7: The constructs for list types in CPL.

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.

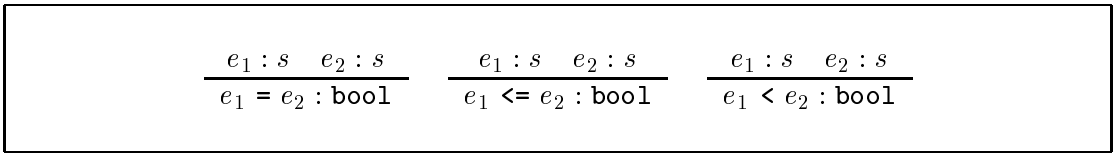

Figure 8: The constructs for comparing objects in CPL.

CPL supports conversion between lists, bags, and sets. These operators, listed in Figure 9, correspond to the monad morphisms mentioned in Wadler [25]. The expression $\texttt{ext}\{e_1 \mid \backslash x^t \texttt{<--}\ e_2\}$ 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 $\texttt{ext}\{\texttt{|}\, e_1 \mid \backslash x^t \texttt{<-}\ e_2 \,\texttt{|}\}$ 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 $\texttt{ext}\{e_1 \mid \backslash x^t \texttt{<---}\ e_2\}$ 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 $\texttt{ext}\{\texttt{|}\, e_1 \mid \backslash x^t \texttt{<---}\ e_2 \,\texttt{|}\}$ 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 $o_1 \texttt{<=} \cdots \texttt{<=}\ o_n$.

CPL has a construct for sending requests to external data servers. This construct is given in Figure

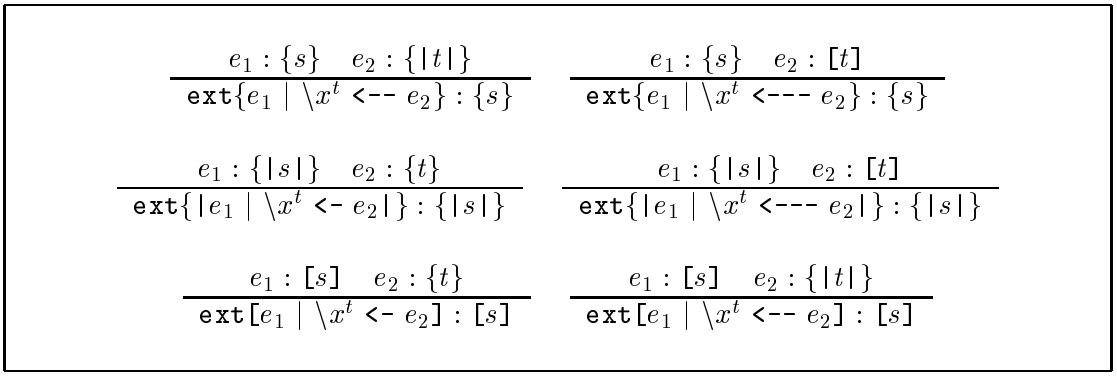

Figure 9: The constructs for list-bag-set interactions in CPL.

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.

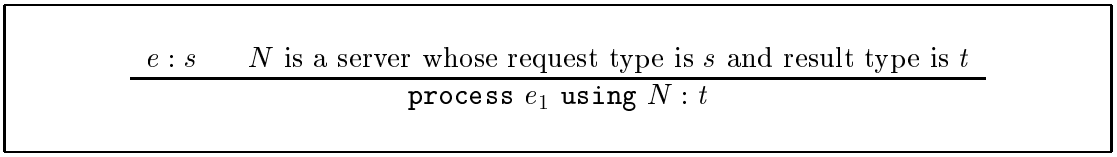

Figure 10: The construct for sending a request to a server in CPL.

CPL supports some syntactic sugar on expressions:

- The expression $e_1(e_2)$ means e_1 `@` e_2, provided e_1 is a function symbol.
- The expression e_1 `.` e_2 means e_2 `@` 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 [16], Remy [18], 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 [11], records, and variants [7] 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 [12] 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 [25]. 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 [23] and Haskell [5]. 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:

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

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 $e_i : \{\!|\, s_i \,|\!\}$.
- A_i is a list-abstraction $\backslash x^{s_i}$ `<---` e_i. Then A_i* is the type-derivation showing e_i : $[s_i]$.

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 [25]. Let us use Δ as a meta notation for a sequence of A_i.

For set comprehensions:

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

- Interpret $\{e' \mid e, \Delta\}$ as `if` e `then` $\{e' \mid \Delta\}$ `else` $\{\}$.

For bag comprehensions:

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

For list comprehensions:

- Interpret `[`$e' \mid \backslash x^s$ `<-` e, Δ`]` as `ext[[`$e' \mid \Delta$`]] |` $\backslash x^s$ `<-` e`]`.
- Interpret `[`$e' \mid \backslash x^s$ `<--` e, Δ`]` as `ext[[`$e' \mid \Delta$`]] |` $\backslash x^s$ `<--` e`]`.
- Interpret `[`$e' \mid \backslash x^s$ `<---` e, Δ`]` as `ext[[`$e' \mid \Delta$`]] |` $\backslash x^s$ `<---` e`]`.
- Interpret `[`$e' \mid e, \Delta$`]` as `if` e `then [`$e' \mid \Delta$`] else []`.

The basic idea of interpreting comprehension in terms of the monad transformation constructs `ext` is due to Wadler [25]. 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 [8], Miranda [24], KRC [23], Id [14], etc. As demonstrated by Trinder [22], 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 [4] and ML [13]. In particular, it supports partial-record patterns found in Machiavelli [15] 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
	|	$\backslash x$	Match anything and bind it to x
	|	$\backslash x$`&`S	Match using S and bind it to x
	|	`(`$l_1 : S_1, \cdots, l_n : S_n$`)`	Match records
	|	`(`$l_1 : S_1, \cdots, l_n : S_n, ...$`)`	Match records partially
	|	`()`	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 `(`$l_1 : S_1, \cdots, l_n : S_n, ...$`)` 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 `(`$S_1, \cdots, S_n$`)` as syntactic sugar for the pattern `(`$\#1 : S_1, \cdots, \#n : S_n$`)`.

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

For case-expressions:

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

- Treat `case` e `of <`l_1 `:` S_1`> =>` e_1 `or` $\cdots$ `or <`l_n `:` S_n`> =>` e_n `otherwise` e' as `case` e `of <` l_1 `:` $\backslash x_1$ `> =>` $(S_1$ `=>` $e_1)$ `@` x_1 `or` $\cdots$ `or <`l_n `:` $\backslash x_n$`> =>` $(S_n$ `=>` $e_n)$ `@` x_n `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 `ext`$\{e_1 \mid S$ `<-` $e_2\}$ as `ext`$\{(S$ `=>` $e_1)$ `@` $x \mid \backslash x$ `<-` $e_2\}$, where x is fresh and S is not ultra-simple.
- Treat `ext`$\{e_1 \mid S$ `<--` $e_2\}$ as `ext`$\{(S$ `=>` $e_1)$ `@` $x \mid \backslash x$ `<--` $e_2\}$, where x is fresh and S is not ultra-simple.
- Treat `ext`$\{e_1 \mid S$ `<---` $e_2\}$ as `ext`$\{(S$ `=>` $e_1)$ `@` $x \mid \backslash x$ `<---` $e_2\}$, 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
	|	$\backslash x$	Match anything and bind it to x
	|	$\backslash x$`&`E	Match using E and bind it to x
	|	`(`$l_1 : E_1, \cdots, l_n : E_n$`)`	Match records
	|	`(`$l_1 : E_1, \cdots, l_n : E_n, ...$`)`	Match records partially
	|	`()`	Match `()` only
	|	c	Match constant c only
	|	`<`$l : E$`>`	Match variants
	|	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 `ext`$\{e_1 \mid E$ `<-` $e_2\}$ as `ext`$\{$`if` $x = A$ `then` e_1 `else` $\{\} \mid E'$ `<-` $e_2\}$, 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 `ext`$\{e_1 \mid E$ `<-` $e_2\}$ as `ext`$\{$`case` x `of` `<`$l : \backslash y$`>` `=>` `ext`$\{e_1 \mid E''$ `<-` $\{y\}\}$ `otherwise` $\{\} \mid E'$ `<-` $e_2\}$, where x and y are fresh, `<`$l : E''$`>` is a subpattern in E and E' is obtained from E by replacing one occurrence of `<`$l : E''$`>` by $\backslash x$.

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

- Treat `ext`$\{e_1 \mid E$ `<---` $e_2\}$ as `ext`$\{$`if` $x = A$ `then` e_1 `else` $\{\} \mid E'$ `<---` $e_2\}$, 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 `ext`$\{e_1 \mid E$ `<---` $e_2\}$ as `ext`$\{$`case` x `of` `<`$l : \backslash y$`>` `=>` `ext`$\{e_1 \mid E''$ `<-` $\{y\}\}$ `otherwise` $\{\} \mid E'$ `<---` $e_2\}$, where x and y are fresh, `<`$l : E''$`>` is a subpattern in E and E' is obtained from E by replacing one occurrence of `<`$l : E''$`>` 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 [25]. 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: x `=>` $\{$`(`x, y`)` $\mid$ `(`x, y`)` `<-` $R\}$. 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: x `=>` $\{$`(`x, y`)` $\mid$ `(`$x, \backslash y$`)` `<-` $R\}$. The CPL program that implements the correct but obscured meaning of the original expression is: x `=>` $\{$`(`x, y`)` $\mid$ `(`$\backslash x, \backslash y$`)` `<-` $R\}$. 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 [19] 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 [16], Remy [18], and Jategaonkar and Mitchell [10]. 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 [21]. 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 [13], 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.)

References

[1] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[2] 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.

[3] 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.

[4] 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.

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

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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 newline, followed by `! beginsml`, followed by a newline. The end of a piece of embedded ML codes is indicated by a newline, followed by `! endsml`, 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 modified from Kyle Hart's original version.

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

Make connections to Sybase servers. You have to supplied 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-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 \verb`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`.

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 `fix`$(f) = f($`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. `handle`(F, H) works like this. First, $F()$ is executed. If some `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 `error`(S) is raised.

- `before: ('1,  '1 -> '2) -> '2`

  This is a sequencing operator. `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. `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-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);
  ```

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);
```

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-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);
  ```

C.5 Numeric Operations

These are implemented in the Kleisli module NUM.

- `+: (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"`.

C.6 String Operations

These are implemented in the Kleisli module 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-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-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-newline: string`

  The newline character.

- `string-quote: string`

  The string quotation character.

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.

Caution. These arrays use a proposed extension to the CPL type system so that we can write `num^k` to denote the type of a tuple of k numbers and `array(num^k, `s`)` to denote the type of k-dimensional arrays with elements of type s.

- `array-mk : (num^k, [''1]) -> array(num^k, ''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(num^k, ''1) -> (num^k, [''1])`

  The inverse of `array-mk`.

- `array-dim : array(num^k, ''1) -> num^k`

  Extract the sizes of the dimensions of the array.

- `array-dimk : (array(num^j, ''1), num) -> num`

  Extract the size of the specified dimension of the array.

- `array-size : array(num^k, ''1) -> num`

  The number of elements in the array.

- `array-checkbound : (array(num^k, ''1), num^k) -> bool`

  Is the index within the bounds of the array?

- `array-jump : (array(num^k, ''1), num) -> ''1`

  View the array as a one-dimensional vector and return the nth element of the vector.

- `array-offset : (array(num^k, ''1), num^k) -> num`

  Given an index into the k-dimensional array, return equivalent index into the equivalent one-dimensional vector.

- `array-sub : (array(num^k, ''1), num^k) -> ''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 : (num^k, num^k -> ''1) -> array(num^k, ''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(num^k,''1),num^k,num^k) -> array(num^k,''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(num^k, ''1) -> {(num^k, ''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 : (num^k, (''1,''2) -> ''2, ''2) -> {(num^k,''1)} -> array(num^k,''2)`

  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 [6] 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+ : (num^k, num^k) -> num^k`

  Componentwise addition of two array indices.

- `array-index- : (num^k, num^k) -> num^k`

  Componentwise subtraction of the second index from the first index.

- `array-indexcheck : (num^k, num^k) -> 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 : (num^k, num^k) -> 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
`##`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 `het-multimatch("`$s_1.\cdots.s_n$`", `H`) =` $\{y \mid$ `\`x `<- het-match("`s_1`", `H`),` $\backslash y$ `<- het-multimatch("`$s_2.\cdots.s_n$`", `x`)`$\}$.

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.
- `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.