Querying an Object-Oriented Database Using CPL *

Susan B. Davidson, Carmem Hara† and Lucian Popa
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Email: susan@central.cis.upenn.edu, {chara, lpopa}@saul.cis.upenn.edu

May 1, 1997

Abstract

The Collection Programming Language (CPL) is based on a complex value model of data, and has successfully been used for querying, transforming and integrating data from a wide variety of structured data sources – relational, ACeDB, and ASN.1 among others. However, since there is no notion of objects and classes in CPL, it cannot adequately model recursive types or inheritance, and hence cannot be used to query object-oriented databases (OODBs). By adding a reference type and four operations to CPL – dereference, method invocation, identity test and class type cast – it is possible to express a large class of interesting "safe" queries against OODBs. As an example of how the extended CPL can be used to query an OODB, we will describe how the extended language has been used as a query interface to Shore databases.

1 Introduction

A vast amount of data currently exists in databases, files formatted according to various data exchange formats, and application programs. Although much of this data is logically inter-related and physically connected over the internet, providing integrated access to such heterogeneous data sources remains elusive. While several interfaces have been developed to provide browsing access to such data, it is a much more challenging problem to provide efficient "bulk" access. "Bulk" access entails the ability to extract, combine and transform data across multiple data sources in one query, and to do so efficiently. Although commercial solutions exist for querying and transforming data across multiple relational databases, the techniques do not extend beyond the "sets of records" type system of relational databases to the more complex types found in data exchange formats, where arbitrary nesting of records, variants, sets, lists, and bags is common. It is therefore difficult, if not impossible, to use a single language or access mechanism to obtain, combine and efficiently transform data from multiple non-relational data sources.

The approach that has been taken by the Kleisli data integration project at the University of Pennsylvania is to use a complex-value model of data, and develop a query language called the Collection Programming Language (CPL) for manipulating such data. By looking at the operations that are naturally associated with each of the data types – records, sets, lists, variants and bags – CPL is able to generalize languages

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO DAAH04-93-G0129, and ARPA N00014-94-1-1086.
†Partially supported by CNPq-Brazil, and Universidade Federal do Paraná.

for nested relations to a complex type system. Moreover, rewrite rules associated with these operations naturally extend many of the algebraic optimization techniques for relational systems to this more complex type system, providing the basis for a powerful query optimizer. Since the type system of CPL subsumes many of those we have encountered in practice – relational databases as well as data exchange formats such as ASN.1 and ACeDB – data can be extracted from multiple heterogeneous data sources and represented internally in Kleisli using the types in which they were stored rather than simplifying them into a "flat" type (as it is done in Tsimmis [11] or Information Manifold [15]). Queries which integrate and combine data across multiple data sources can then be optimized using this type information, dramatically improving the performance of the Kleisli data integration system.

However, Kleisli has not been connected to object-oriented databases (OODBs) until now, since CPL does not support classes, recursive values, methods and inheritance. In order to capture this behavior in CPL, we therefore extend its type system with a *class type*. References are values of class types and they are associated with a record of attributes and a record of methods, corresponding to the attributes and methods defined for the class to which a reference belongs in the OODB. The *class type* supports only four operations: *dereference, method invocation, identity test, and class type cast.* Through examples, we show that with these simple constructs it is possible to express a large class of interesting queries against OODBs.

A nice side-effect of this extension to CPL is that it can be used as a high-level query language to OODBs that currently provide low-level access in the form of library calls embedded in some host programming language such as C++ or Smalltalk. As an example of this, we have connected Kleisli using the extended CPL language with Shore [9], an OODB under development at the University of Wisconsin. The implementation of a Shore application involves the definition of the schema, its translation to a language for which a binding is offered (currently only C++), and implementation of the methods. From CPL, we associate each object in the database with a *reference*, and access the attributes and methods from the object using the operations defined on the new data type. The resulting language resembles other query languages proposed for OODBs, such as OQL, the query language from the Object Database Standard (ODMG) [10].

The rest of the paper is organized as follows. We start in Section 2 by describing the complex value model used in CPL, and giving the flavor of the language through examples. In Section 3 we describe the extended "object" model. Extensions to CPL to support this new model and examples of queries expressible in extended CPL are presented in Section 4. Section 5 describes implementation aspects of how we connected extended CPL to Shore using Kleisli. Section 6 compares our language to OQL and concludes the paper.

2 CPL: A Query Language for Collection types

The language CPL (Collection Programming Language) is based on a type system that allows arbitrary nesting of the collection types – set, bag and list – together with record and variant types. The types are given by the syntax

$$\tau := \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \mid \mathsf{unit} \mid \ldots \mid \{\tau\} \mid \{|\tau|\} \mid \{\!|\tau|\!\} \mid \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle \mid [a_1 : \tau_1, \ldots, a_n : \tau_n]$$

Here, bool | int | string | unit | ... are the (built-in) base types. The other types are all *constructors* and build new types from existing types. $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ constructs record types from the types $\tau_1, \ldots, \tau_n$. $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$ constructs variant types from the types $\tau_1, \ldots, \tau_n$. $\{\tau\}$, $\{|\tau|\}$, and $\{\!|\tau|\!\}$ respectively construct set, bag, and list types from the type τ. An example of this type system is

```
Student = {[name; string,
            id: int,
            major: <cis: unit, ee: unit>,
            courses: {|[number: int,
                        instructor: string]|}]}
```

Note that a Student can have a major that is either cis or ee, and that their courses are organized as a list of records.

The syntax for values in CPL is a subset of the language that constructs values: $[l_1 = e_1, \ldots, l_n = e_n]$ for records; $<l = e>$ for variants, $\{e_1 \ldots e_n\}$ for sets; and similarly for multisets and lists. For example, a fragment of data conforming to the Student type is

```
{[name="Jane",
  id=9580027,
  major =<cis=()>,
  courses={|[number=500,
             instructor="Peter"],
            [number=550,
             instructor="Susan"]|}]...}
```

This example shows just one member of a set of student records. Since a relation is a set of records, it is also straightforward to represent a relational database in this format. In fact, the type system of CPL (which is slightly larger than the description given here) allows us to express most common data formats.

The language CPL. The syntax of CPL resembles, very roughly, that of relational calculus. However, there are important differences that make it possible to deal with the richer variety of types we have mentioned and to allow function definition within the language. The important syntactic unit of CPL is the *comprehension*, which can be used with a variety of collection types.

As an example of a set comprehension, this is a simple CPL query that extracts the id and courses from a database DB of the type Student

```
{[id = s.id, courses = s.courses]| \s <- DB}
```

Note the use of \s to introduce the variable s. The effect of \s <- DB is to bind s to each element of the set DB. This form is called a *generator*. Each collection type in CPL has to be used with the correspondent generator: <- for sets, <-- for bags, and <- - - for lists. The use of explicit variable binding is needed if we are to use database queries in conjunction with function definition or *pattern matching* as in the example below, which is equivalent to the one above. Note that the ellipsis "..." matches any remaining fields in the DB record.

```
{[id = i, courses = c]| [id = \i, courses = \c, ...] <- DB}
```

Apart from the fact that the queries above return a nested structure, they can be readily expressed in relational calculus. The following queries perform simple restructuring:

```
{[id = i, course = c.number]| [id = \i, courses = \cc, ...] <- DB, \c <- - - cc}

{[course = c, students = {x.id| \x <- DB, [number = c, ...] <- - - x.courses }]|
    \y <- DB, [number = \c, ...] <- - - y.courses }
```

Note the use of a list generator (<- - -) for courses. When there is a generator inside a comprehension of a different type, the collection on which the generator iterates is converted to the type of the comprehension. In this case, courses is converted to a set before the iteration. The first query "flattens" the nested relation; the second restructures it so that the database becomes a database of courses with associated students. Operations such as these can be expressed in nested relational algebra and in certain object-oriented query languages. The strength of CPL is that it has more general collection types, allows function definition and can also exploit variants, which may be used in pattern matching:

```
{[id = i, name = n ]| [id = \i, name = \n, major = <cis = ()>, ... ] <- DB }
```

This gives us the id and name of students whose major is "cis".

The syntax of functions is given by $\backslash x \Rightarrow e$, where e is an expression that may contain the variable x. We can give this function (or any other CPL expression) a name with the syntax define $f == e$ which causes f to act as a synonym for the expression e. Thus, the number of courses in which a given student is enrolled in can be expressed as the function

define courses_of == \x ⇒{c.number| [name = x, courses = \cc, ...] <− DB, c <− − − cc}

These examples illustrate part of the expressive power of CPL. A more detailed description of the language is given in [6]; see also [16] for a discussion of adding arrays. An important property of comprehension syntax is that it is derived from a more powerful programming paradigm on collection types, that of *structural recursion* [3, 4]. This more general form of computation on collections allows the expression of aggregate functions such as summation, as well as functions such as transitive closure, that cannot be expressed through comprehensions alone. However, if a fixed number of aggregate operations are added to CPL as primitives, then the language parallels the expressive power of SQL, since CPL restricted to input and output to be flat relation types expresses the relational algebra [6]. The advantage of using comprehensions is that they have a well-understood set of transformation rules [22, 19, 20] that generalize many of the known optimizations of relational query languages to work for a complex value type system.

3 A Data Model with Object Identities

In general, each object in a Object Oriented Data Base (OODB) has a unique identifier (OID) which is assigned by the Database Management System at the object creation time. The value of the OID usually does not depend on the value of the properties of the object, but it is some number generated by the system. Objects can refer to each other using OIDs. For example, student objects can "point" to course objects, and vice-versa. This reference mechanism permits the definition of recursive data structures, which can not be directly represented by CPL types as described in the previous section.

In order to query objects stored in a OODB from CPL, we extended its type system with a new type called *class*. Values of this type correspond to OIDs in the OODB. In the following section we present our data model, which is an extension of [14] with methods.

3.1 Types and schemas

A class has a name, and it is associated with a record type of attributes, a record type of methods, and a set of superclasses. Attributes represent properties of a given class. In the record type of attributes, the type of each field can be a class or any CPL type presented in Section 2. For simplicity, in what follows only sets, records, variants, and classes are treated; bags and lists can be handled similarly.

Let $\mathcal{C}$ be a finite set of class names, ranged over by $C, C', \ldots$, and $\mathcal{A}$ be a fixed countable set of labels, ranged over by $a_1, a_2, \ldots, m_1, m_2, \ldots$.

We define the data types ($\mathbf{Types}^{\mathcal{C}}$) and the method types ($\mathbf{MTypes}^{\mathcal{C}}$) as follows:

- $\mathbf{Types}^{\mathcal{C}}$

  $\tau ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \mid \mathsf{unit} \mid \ldots \mid \{\tau\} \mid \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle \mid [a_1 : \tau_1, \ldots, a_n : \tau_n] \mid C$

- $\mathbf{MTypes}^{\mathcal{C}}$

$$\mu ::= [m_1 : C \times \tau_1 \rightarrow \tau_1', \ldots, m_n : C \times \tau_n \rightarrow \tau_n']$$

Base types ($\underline{b}$), sets ($\{\tau\}$), records ($[a_1 : \tau_1, \ldots, a_n : \tau_n]$), and variants ($<a_1 : \tau_1, \ldots, a_n : \tau_n>$) were introduced in Section 2. C is the class type.

A method type is a record where each field is a function with two arguments. The first argument is a value of the class type and corresponds to the object to which the method is applied. This is often called the "self" parameter. The second argument corresponds to the parameters of the method.

We can now define a **database schema** as a quadruple $(\mathcal{C}, \mathcal{S}, \mathcal{M}, \preceq)$, where

- $\mathcal{C}$ is a finite set of classes
- $\mathcal{S}$ is a schema mapping $\mathcal{S} : \mathcal{C} \rightarrow \mathbf{Types}^{\mathcal{C}}$, such that for any $C \in \mathcal{C}$, $C \stackrel{\mathcal{S}}{\mapsto} \tau^C$, where τ^C is a record type, with fields that correspond to the attributes of the class.
- $\mathcal{M}$ is a method mapping $\mathcal{M} : \mathcal{C} \rightarrow \mathbf{MTypes}^{\mathcal{C}}$
- $\preceq$ is a partial order on $\mathcal{C}$, such that
  - For each pair C, C' in $\mathcal{C}$, if $C \preceq C'$ and

$$\begin{aligned} \mathcal{S}(C') &\equiv [a_1 : \tau_1, \ldots, a_m : \tau_m] \\ \mathcal{M}(C') &\equiv [m_1 : C' \times \tau_1 \rightarrow \tau_1', \ldots, m_k : C' \times \tau_k \rightarrow \tau_k'] \end{aligned}$$

    then

$$\begin{aligned} \mathcal{S}(C) &\equiv [a_1 : \tau_1, \ldots, a_m : \tau_m, \ldots, a_n : \tau_n] \\ \mathcal{M}(C) &\equiv [m_1 : C \times \tau_1 \rightarrow \tau_1', \ldots, m_k : C \times \tau_k \rightarrow \tau_k', \ldots, m_l : C \times \tau_l \rightarrow \tau_l'] \end{aligned}$$

    where $m \leq n$, $k \leq l$, and $a_1, \ldots, a_n$, $m_1, \ldots, m_l$ are distinct label names.
  - Let $Label : \mathcal{C} \rightarrow \mathcal{A}$ be a function that maps each class to the set of attribute and method labels in $\mathcal{S}(C)$, and $\mathcal{M}(C)$. If C is a subclass of C' and C'', and $Label(C') \bigcap Label(C'') \neq \emptyset$, then there is a class C''', superclass of C, C', and C'', such that $Label(C''') = Label(C') \bigcap Label(C'')$.

Informally, $\preceq$ defines a class hierarchy where subclasses can only provide additional attributes and methods to their superclasses. If $C \preceq C_1$, and $C \preceq C_2$, and an attribute or method l exists in both C_1, and C_2, then there must exist C', $C_1 \preceq C'$, $C_2 \preceq C'$, such that l is defined in C'.

Example. Let us consider a school database with the set of classes given by

$$\mathcal{C} \equiv \{\mathsf{Person}, \mathsf{Student}, \mathsf{Course}\}$$

The schema mapping is given by

$$\begin{aligned} \mathcal{S}(\mathsf{Person}) &\equiv [\mathsf{name} : \mathsf{string}, \mathsf{age} : \mathsf{int}, \mathsf{mother} : <\mathsf{none} : \mathsf{unit}, \mathsf{some} : \mathsf{Person}>] \\ \mathcal{S}(\mathsf{Student}) &\equiv [\mathsf{name} : \mathsf{string}, \mathsf{age} : \mathsf{int}, \mathsf{mother} : <\mathsf{none} : \mathsf{unit}, \mathsf{some} : \mathsf{Person}>, \mathsf{enrolled\_in} : \{\mathsf{Course}\}] \\ \mathcal{S}(\mathsf{Course}) &\equiv [\mathsf{number} : \mathsf{int}, \mathsf{TA} : <\mathsf{none} : \mathsf{unit}, \mathsf{some} : \mathsf{Student}>, \mathsf{enrolls} : \{\mathsf{Student}\}] \end{aligned}$$

The method mapping is given by

$$\begin{aligned} \mathcal{M}(\mathsf{Person}) &\equiv [] \\ \mathcal{M}(\mathsf{Student}) &\equiv [\mathsf{num\_of\_courses} : \mathsf{Student} \times \mathsf{unit} \rightarrow \mathsf{int}] \\ \mathcal{M}(\mathsf{Course}) &\equiv [\mathsf{num\_of\_students} : \mathsf{Course} \times \mathsf{unit} \rightarrow \mathsf{int}] \end{aligned}$$

And the only subclass relationship we have is

$$\mathsf{Student} \preceq \mathsf{Person}$$

3.2 Database Instances

Domain of database values. Values of class types are object identities. We define an *object-identity assignment* $\sigma^{\mathcal{C}}$ for a set of classes $\mathcal{C}$ as a mapping from $\mathcal{C}$ to a family of disjoint finite sets of object identities. I.e., $C \overset{\sigma^{\mathcal{C}}}{\mapsto} \sigma^C$, such that if $C \neq C'$ then $\sigma^C \bigcap \sigma^{C'} = \emptyset$. Let us denote by $\mathbf{D}^{\underline{b}}$ the domain of the base type $\underline{b}$, for any $\underline{b}$.

The domain of our model $\mathbf{D}(\sigma^{\mathcal{C}})$ is defined as the least set satisfying the equation:

$$\mathbf{D}(\sigma^{\mathcal{C}}) \equiv (\bigcup_{\underline{b}} \mathbf{D}^{\underline{b}}) \cup (\bigcup_{C \in \mathcal{C}} \sigma^C) \cup (\mathcal{A} \xrightarrow{\sim} \mathbf{D}(\sigma^{\mathcal{C}})) \cup (\mathcal{A} \times \mathbf{D}(\sigma^{\mathcal{C}})) \cup P_{fin}(\mathbf{D}(\sigma^{\mathcal{C}})) \cup$$

$$(\bigcup_{C \in \mathcal{C}} (\mathcal{A} \xrightarrow{\sim} (\sigma^C \times \mathbf{D}(\sigma^{\mathcal{C}}) \Rightarrow \mathbf{D}(\sigma^{\mathcal{C}}))))$$

where $A \Rightarrow B$ denotes the set of all functions with domain A and codomain B, and $A \xrightarrow{\sim} B$ denotes the set of partial functions from A to B with finite domains.

Denotations of types. Given a schema $(\mathcal{C}, \mathcal{S}, \mathcal{M}, \preceq)$, and an object identity assignment $\sigma^{\mathcal{C}}$, the interpretation of each type τ in $\mathbf{Types}^{\mathcal{C}}$ and $\mathbf{MTypes}^{\mathcal{C}}$, $[\![\tau]\!]\sigma^{\mathcal{C}}$, is defined by

$$\begin{aligned}
[\![\underline{b}]\!]\sigma^{\mathcal{C}} &\equiv \mathbf{D}^{\underline{b}} \\
[\![C]\!]\sigma^{\mathcal{C}} &\equiv \bigcup_{C' \preceq C} \sigma^{C'} \\
[\![\{\tau\}]\!]\sigma^{\mathcal{C}} &\equiv P_{fin}([\![\tau]\!]\sigma^{\mathcal{C}}) \\
[\![\texttt{[}a_1 : \tau_1, \ldots, a_n : \tau_n\texttt{]}]\!]\sigma^{\mathcal{C}} &\equiv \{f \in \mathcal{A} \xrightarrow{\sim} \mathbf{D}(\sigma^{\mathcal{C}}) \mid dom(f) = \{a_1, \ldots, a_n\} \\
&\quad \textit{and } f(a_i) \in [\![\tau_i]\!]\sigma^{\mathcal{C}}, i = 1, \ldots, n\} \\
[\![\texttt{<}a_1 : \tau_1, \ldots, a_n : \tau_n\texttt{>}]\!]\sigma^{\mathcal{C}} &\equiv (\{a_1\} \times [\![\tau_1]\!]\sigma^{\mathcal{C}}) \cup \ldots \cup (\{a_n\} \times [\![\tau_n]\!]\sigma^{\mathcal{C}}) \\
[\![\texttt{[}m_1 : C \times \tau_1 \rightarrow \tau_1', \ldots, m_n : C \times \tau_n \rightarrow \tau_n'\texttt{]}]\!]\sigma^{\mathcal{C}} &\equiv \bigcup_{C \in \mathcal{C}} \{f \in \mathcal{A} \xrightarrow{\sim} (\sigma^C \times \mathbf{D}(\sigma^{\mathcal{C}}) \Rightarrow \mathbf{D}(\sigma^{\mathcal{C}})) \mid \\
&\quad dom(f) = \{m_1, \ldots, m_n\} \textit{ and} \\
&\quad f(m_i) \in (\sigma^C \times [\![\tau_i]\!]\sigma^{\mathcal{C}}) \Rightarrow [\![\tau_i']\!]\sigma^{\mathcal{C}}, i = 1, \ldots, n\}
\end{aligned}$$

Note that the domain of a class type includes the object identities of the class and all its subclasses. That is, $[\![C]\!]\sigma^{\mathcal{C}} \subseteq [\![C']\!]\sigma^{\mathcal{C}}$ whenever $C \preceq C'$. This formalizes the fact that an object of a subclass C can be viewed also as an object of a superclass C' of C.

Database instance. A database instance of a schema $\mathcal{S}$ is a quadruple $\mathcal{I} = (\sigma^{\mathcal{C}}, classOf, \nu^{\mathcal{C}}, \psi^{\mathcal{C}})$, where

- $\sigma^{\mathcal{C}}$ is an object identity assignment
- $classOf : \bigcup_{C \in \mathcal{C}} \sigma^C \rightarrow \mathcal{C}$, is a function that maps each object identity o to a class C such that $o \in \sigma^C$.
- $\nu^{\mathcal{C}}$ is a family of functions $\nu^C : \sigma^C \rightarrow [\![\mathcal{S}(C)]\!]\sigma^{\mathcal{C}}$, and
- ψ is a function $\psi : \mathcal{C} \rightarrow \bigcup_{\mathcal{C} \in \mathcal{C}} [\![\mathcal{M}(\mathcal{C})]\!]\sigma^{\mathcal{C}}$, such that $\psi(C) \in [\![\mathcal{M}(C)]\!]\sigma^{\mathcal{C}}$.

The object identity assignment function associates disjoint sets of identities to each class. The function $classOf$ is its inverse, i.e., from each object identity it returns its class. For each class C, ν^C associates a record of attribute values to each object in σ^C. As opposed to the value assignment functions $\nu^{\mathcal{C}}$ that

associates objects with values, $\psi^{\mathcal{C}}$ associates classes with functions that implement their methods. In our model, we assume that methods have no side effects, that is, methods do not update the database. Note that the above definition allows method overloading, since it's possible to assign a different function to a method inherited from a superclass.

Example. An instance of the school database introduced in the previous section follows.

Our object identities are:

$$\begin{aligned} \sigma^{\mathsf{Person}} &\equiv \{\mathsf{johnOid}\} \\ \sigma^{\mathsf{Student}} &\equiv \{\mathsf{maryOid}, \mathsf{joeOid}, \mathsf{saraOid}\} \\ \sigma^{\mathsf{Course}} &\equiv \{\mathsf{500Oid}, \mathsf{501Oid}\} \end{aligned}$$

and the $classOf$ function is defined as

$$\begin{aligned} &classOf(\mathsf{johnOid}) = \mathsf{Person} \\ &classOf(\mathsf{maryOid}) = classOf(\mathsf{joeOid}) = classOf(\mathsf{saraOid}) = \mathsf{Student} \\ &classOf(\mathsf{500Oid}) = classOf(\mathsf{501Oid}) = \mathsf{Course} \end{aligned}$$

The attribute value assignment are:

$$\begin{aligned} \nu^{\mathsf{Person}}(\mathsf{johnOid}) &\equiv [\mathsf{name} \mapsto'' \mathsf{john}'', \mathsf{age} \mapsto 5, \mathsf{mother} \mapsto \langle \mathsf{some} : \mathsf{saraOid} \rangle] \\ \nu^{\mathsf{Student}}(\mathsf{maryOid}) &\equiv [\mathsf{name} \mapsto'' \mathsf{mary}'', \mathsf{age} \mapsto 20, \mathsf{mother} \mapsto \langle \mathsf{none} : \mathsf{unit} \rangle, \mathsf{enrolled\_in} \mapsto \{\mathsf{500Oid}, \mathsf{501Oid}\}] \\ \nu^{\mathsf{Student}}(\mathsf{joeOid}) &\equiv [\mathsf{name} \mapsto'' \mathsf{joe}'', \mathsf{age} \mapsto 22, \mathsf{mother} \mapsto \langle \mathsf{none} : \mathsf{unit} \rangle, \mathsf{enrolled\_in} \mapsto \{\mathsf{500Oid}\}] \\ \nu^{\mathsf{Student}}(\mathsf{saraOid}) &\equiv [\mathsf{name} \mapsto'' \mathsf{sara}'', \mathsf{age} \mapsto 26, \mathsf{mother} \mapsto \langle \mathsf{none} : \mathsf{unit} \rangle, \mathsf{enrolled\_in} \mapsto \{\mathsf{501Oid}\}] \\ \nu^{\mathsf{Course}}(\mathsf{500Oid}) &\equiv [\mathsf{number} \mapsto 500, \mathsf{TA} \mapsto \langle \mathsf{some} : \mathsf{saraOid} \rangle, \mathsf{enrolls} \mapsto \{\mathsf{maryOid}, \mathsf{joeOid}\}] \\ \nu^{\mathsf{Course}}(\mathsf{501Oid}) &\equiv [\mathsf{number} \mapsto 501, \mathsf{TA} \mapsto \langle \mathsf{none} : \mathsf{unit} \rangle, \mathsf{enrolls} \mapsto \{\mathsf{maryOid}, \mathsf{saraOid}\}] \end{aligned}$$

In our model, the semantics of methods are given by external functions defined by the OODB that contains the objects. Method assignments only make methods refer to such external functions, as in

$$\psi(\mathsf{Student}) \equiv [\mathsf{num\_of\_courses} \mapsto \textit{a function that implements it}]$$

4 Extending CPL with Objects

In this section we extend CPL with operations for class types. References are values of class types and correspond to OIDs in a OODB. Using examples, we will show that with only four new constructs for this type, a number of interesting queries can be expressed in the language. The examples use the School database defined in the previous section.

4.1 The Extended Language

The syntax and typing rules of the constructs for class types are illustrated in Figure 1. The typing rules for the constructs of the other types can be found in [22] and are given in Appendix A.

For each class $C \in \mathcal{C}$, we define a constant ext_C, corresponding to the set of object identities from class C and all subclasses of C. This is usually called the extent of a class. For example, ext_Person is the union of OID's from class Person and Student.

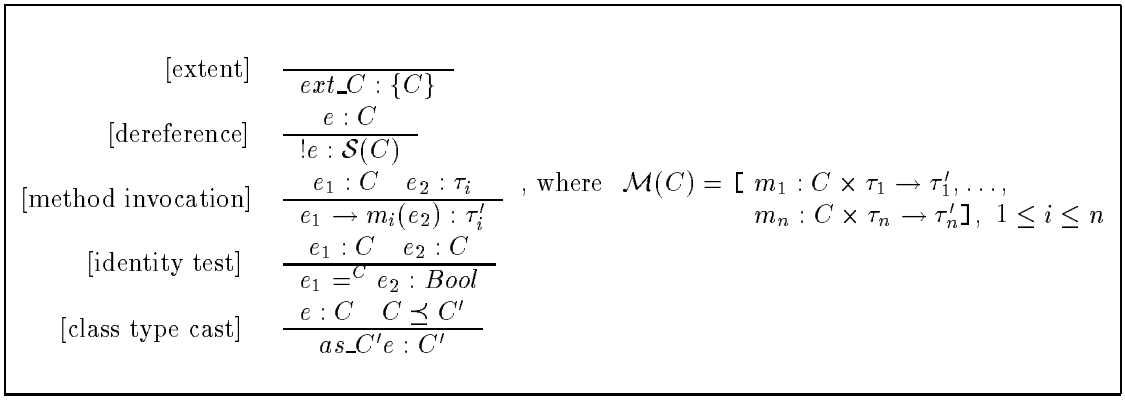

Figure 1: The constructs for class types in CPL

The dereference operation returns a record with the attributes of an object. As an example, the following query returns the name, age, and mother of all persons in ext_Person.

```
{!x|  \x  <- ext_Person}
```

To extract only some attributes of objects, we can combine the dereference operation with record projection, as shown in the next query that returns only the name of all persons.

```
{!x.name|  \x  <- ext_Person}
```

To call a function defined for a class we use the method invocation operation. The following query gets the courses with more than fifteen students enrolled.

```
{!c.number | \c  <- ext_Course, c->num_of_students()> 15 }
```

The identity test executed on objects o_1, and o_2 returns true if both have the same OID. Structural equality on objects can be implemented by first dereferencing them and then testing equality on the resulting records. This implements "shallow structural test", because if the record contains a reference, then the test on references is the identity test. The example below returns the courses in which both "mary" and "joe" are enrolled. Note that if there are two courses with exactly the same value for all the attributes, and "mary" is enrolled in one of them and "joe" in the other, this course is not part of the result since we are testing equality on the identity of the objects.

```
{ !maryCourse.number |
 \m  <- ext_Student, !m.name = "mary", \maryCourse  <- !m.enrolled_in,
 \j  <- ext_Student, !j.name = "joe", \joeCourse  <- !j.enrolled_in,
 maryCourse = joeCourse }
```

A class type cast operation, as_C, is defined for each class $C \in \mathcal{C}$. When it is applied to an object e of the class type C', $C' \preceq C$, it changes the type of e to C. As a consequence, attributes and methods defined in C' which do not exist in C are hidden. Note that this operation is only well defined for an object of a subclass to be transformed to an object of a superclass. The class type cast operation is the only form of subtyping in the extended CPL. This is explicit subtyping as opposed to the implicit subtyping that is common in OO languages like Java or C++.

The following example gives the name of all persons who are not students. Note that it would not type check without the application of the as_Person operation, since the identity test, used to implement set membership, is only well defined for objects from the same class.

```
{ !p.name | \p <- ext_Person, p not in {as_Person(s) | \s <- ext_Student }}
```

We also extend the language to use objects in pattern matching. Using the fact that CPL already supports pattern matching on records, and that the result of a dereference operation is a record type, we interpret the query

$\{e_1 |$ ref of C p $\leftarrow e_2\}$

as the set of elements e_1 such that the dereference of an element of the set e_2 of type $\{C\}$ matches the record pattern p.

This is illustrated in the example below, which returns the course numbers in ext_Course that have a TA and the name of the TA.

```
{ [number=n, TA=!s.name] | ref of Course[number = \n, TA = <some = \s>, ...] <- ext_Course }
```

The following syntactic sugar was also added: the expression $e \rightarrow l$ means $(!e).l$ (dereference of e followed by selection of l), where l is a label of an attribute of e. This is convenient because we use the same piece of syntax ($\rightarrow$) both to invoke a method and to get the value of an attribute from an object. The query below gets the numbers of courses in which there is at least one mother enrolled. It illustrates the syntactic sugar described and also the use of layered patterns \x &P. If the entire pattern P matches, then the value that matches P is also bound to the variable x. This value is viewed both through the pattern and as a whole.

```
{c -> number | \s &ref of Student[enrolled_in: \x, ...] <- ext_Student, \c <- x,
               as_Person(s) in {m | ref of Person[mother: <some: \m>, ...] <- ext_Person }}
```

Note that it is not possible to create objects in the language, since there is no constructor for class types. Values of this type represent objects stored in a database that were brought into the system as a result of function calls executed against the database. Also, there is no assignment on references.

4.2 The Semantics of the Language

This section defines the semantics of the constructs introduced in the previous section. We first present some additional definitions.

- **Coercion functions**. For each pair of classes C, C' in $\mathcal{C}$ such that $C' \preceq C$, we define a coercion function $f^{C' \rightarrow C}$ such that

$$f^{C' \rightarrow C} : [\![\mathcal{S}(C')]\!]\sigma^{\mathcal{C}} \rightarrow [\![\mathcal{S}(C)]\!]\sigma^{\mathcal{C}}$$

  Given x in $[\![\mathcal{S}(C')]\!]\sigma^{\mathcal{C}}$, such that $dom(x) = \{a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+l}\}$, $dom(f^{C' \rightarrow C}(x)) = \{a_1, \ldots, a_{k+l}\}$, and $f^{C' \rightarrow C}(x) = x|_{\{a_1, \ldots, a_k\}}$.

  That is, given a record x with the attributes of an object o, $o \in \sigma^{C'}$, $f^{C' \rightarrow C}(x)$ hides the additional attributes defined for C', which are not defined for C, resulting in a record with only the attributes defined for C. Note that this is a semantic function, defined to give the interpretation of the operations in our query language. It is not an operation in the query language.

  As an example of the use of the coercion function we have

$V[\![ext\_C]\!]\mathcal{I} \equiv \bigcup_{C' \preceq C} \sigma^{C'}$
$V[\![as\_C\ e]\!]\mathcal{I} \equiv V[\![e]\!]\mathcal{I}$, where $e : C'$
$V[\![!e]\!]\mathcal{I} \equiv f^{C' \to C}(\nu^{C'}(V[\![e]\!]\mathcal{I}))$, where $C' = classOf(V[\![e]\!]\mathcal{I})$, and $e : C$.
$V[\![e_1 \to a_i(e_2)]\!]\mathcal{I} \equiv (\Pi^{C'}_{a_i}(\psi(C'))(V[\![e_1]\!]\mathcal{I}, V[\![e_2]\!]\mathcal{I})$, where $C' = classOf(V[\![e_1]\!]\mathcal{I})$, and $e_1 : C$.
$V[\![e_1 =^C e_2]\!]\mathcal{I} \equiv \begin{cases} \mathbf{T}, & \text{if } V[\![e_1]\!]\mathcal{I} = V[\![e_2]\!]\mathcal{I} \\ \mathbf{F}, & \text{otherwise} \end{cases}$

Figure 2: Semantics of class types constructs

$$f^{Student \to Person}([name \mapsto'' joe'', age \mapsto 22, mother \mapsto \langle none : unit \rangle, enrolled\_in \mapsto \{500Oid\}]) = [name \mapsto'' joe'', age \mapsto 22, mother \mapsto \langle none : unit \rangle]$$

- **Method selection**. Assume for each $C \in \mathcal{C}$ the following mappings for method selection:

$$\Pi^C_{m_i} : [\![\mathcal{M}(C)]\!]\sigma^{\mathcal{C}} \to (\sigma^C \times [\![\tau_i]\!]\sigma^{\mathcal{C}} \Rightarrow [\![\tau_i']\!]\sigma^{\mathcal{C}})$$

where $\mathcal{M}(C) = [m_1 : C \times \tau_1 \to \tau_1', \ldots, m_n : C \times \tau_n \to \tau_n']$, $i = 1, \ldots, n$.

Given a database instance $\mathcal{I} = (\sigma^{\mathcal{C}}, classOf, \nu^{\mathcal{C}}, \psi^{\mathcal{C}})$, the semantic function $V[\![.]\!]\mathcal{I}$ maps expressions of the language to $\mathbf{D}(\sigma^{\mathcal{C}})$. The interpretation of class types constructs is defined in Figure 2. The semantic function on the operations of the other types in the language is defined in Appendix A.

The interpretation of ext_C' tells us that an object o from class C' (that is, $o \in \sigma^{C'}$) is not only an element of ext_C', but also an element of ext_C, for all C that are superclasses of C'. Therefore, o can have different types, depending on which extent it is extracted from. Note, however, that a given expression in the language has always a unique type, since the type system does not have a subsumption rule. The extent inclusion is what captures the hierarchy of classes in our data model.

If an object has type C and we want it to be "seen" as an instance of one of its superclasses, then this has to be explicitly defined using the cast operation. This operation does not change the interpretation of the object, as defined in Figure 2. In another words, the interpretation of an object is always its OID, which is an element of only one set σ^C, since the object-identity assignment $\sigma^{\mathcal{C}}$ maps classes to disjoint sets of OIDs. Changing the type of an object using the cast operation affects the interpretation of other operations that can be applied to it.

The meaning of applying the dereference operation on an object o, $o \in \sigma^{C'}$, involves getting all the attributes defined for object o using the $\nu^{C'}$ function; then, if the type of o in the context is C, and C is a superclass of C', we use the coercion function $f^{C' \to C}$ to hide those attributes not defined for class C.

For method invocation, given an object o, $o \in \sigma^C$, we obtain the function associated with the method by applying ψ on C, which gets a record of functions, and then select the desired method. Note that the function ψ returns the methods associated with class C. Therefore, even if o is statically typed as of type C', $C \preceq C'$, the resolution of the method is dynamic. That is, the function that implements the method to be called is determined by the "most specific class" of the object.

The interpretation of the identity test shows that the equality test on values of a class type is defined on OIDs, instead of structural equality.

4.3 Optimizations

This section presents some initial thoughts on optimizations for reference operations. Since queries in OODBs often involve navigation of the database expressed by path expresssions, a main concern of optimizations proposed in the literature is to minimize the amount of I/O performed by these expressions. In our language, a path expression is defined by a sequence of dereference and method invocation operations, and many of the results should follow through to this environment.

Factorization of common subexpressions. It is common for an object to be dereferenced more than once as different fields are selected in an expression. Since by dereferencing an object all its fields are extracted, it's desirable that the query optimizer executes common subexpression factorization. This optimization would avoid not only dereferencing an object multiple times, but it would also avoid invoking a method or extracting the extent of a class more than once in the same query[1].

Type Cast. A simple optimization on the class type cast is to reduce two applications of the operation to one. That is, we can rewrite $as\_C(as\_C'e)$ to as_Ce, where C' is a subclass of C. The cast operation is very cheap and therefore this optimization does not produce a great impact in terms of performance. The identity test operation is also very simple because it does not require any access to the database.

Extents. If the notion of extents is supported by the underlying database (as, for example, in O_2 [13]), and all classes have an associated extent, the constants ext_C simply require an access to their values. On the other hand, if extents are not maintained by the database (as is the case in Shore [9]), creating them incurs an enormous cost in scanning the database. A possible optimization would be to keep a local copy of the extents in the system in order to avoid scanning the database multiple times in the same session. However, this approach could result in an inconsistent state since the extent is not automatically updated. The local copy also cannot be considered a materialized view of the database, since only OIDs are maintained. The associated values of objects are always accessed from the database, and this can cause problems when it is updated. For example, if a new object o_1 is created and a pre-existing object o_2 is updated to refer to it, o_1 is accessible through o_2 in CPL, but it is not in the extent of o_1's class. It is then unsafe to use such optimization, unless an update mechanism on the local extent copy is implemented.

Indexes and inverse relationships. Other optimization technique that consider inverse relationships and indexes can be performed if this typed information is known to the system. Consider the following example that returns the name of all students of course number 500.

```
{ !s.name | \s <- ext_Student, \c <- !s.enrolled_in, !c.number = 500 }
```

If the system knows that there is an index over number on ext_Course and also that enrolls is the inverse of enrolled_in, the query could be rewritten as

```
{ !s.name | \c <- ext_Course, !c.number = 500, \s <- !c.enrolls }
```

Another possible optimization involve rewriting queries based on the physical organization of objects in files. A general framework for algebraic optimizations and "type-based" optimizations as is the above is described in [12]; algebraic optimizations that exploit access paths can be found in [2].

[1] We consider that each query is executed within a single transaction to guarantee that if a query accesses an object multiple times, the value returned by these operations coincide.

5 The CPL-Shore Interface

CPL is implemented on top of Kleisli, an extensible query system written entirely in ML [17]. Routines within Kleisli manage optimization, query evaluation, and I/O from remote and local data sources. It emphasizes openness: new primitives, optimization rules, data scanners, and data writers can be dynamically introduced into the system.

This openness allowed the integration of a number of data sources to the system [7], including conventional databases, like Sybase, and structured files, like ASN.1, and AceDB. The interface between Kleisli and these data sources is performed by data drivers. Once they are registered in Kleisli, they can be used as primitives in CPL to access the data sources. They perform the task of logging into a specific data source, sending queries in the native form for that source, returning results to Kleisli in internal Kleisli value syntax, and logging out from the data source when the CPL session terminates. The overall architecture of the system is shown in Figure 3.

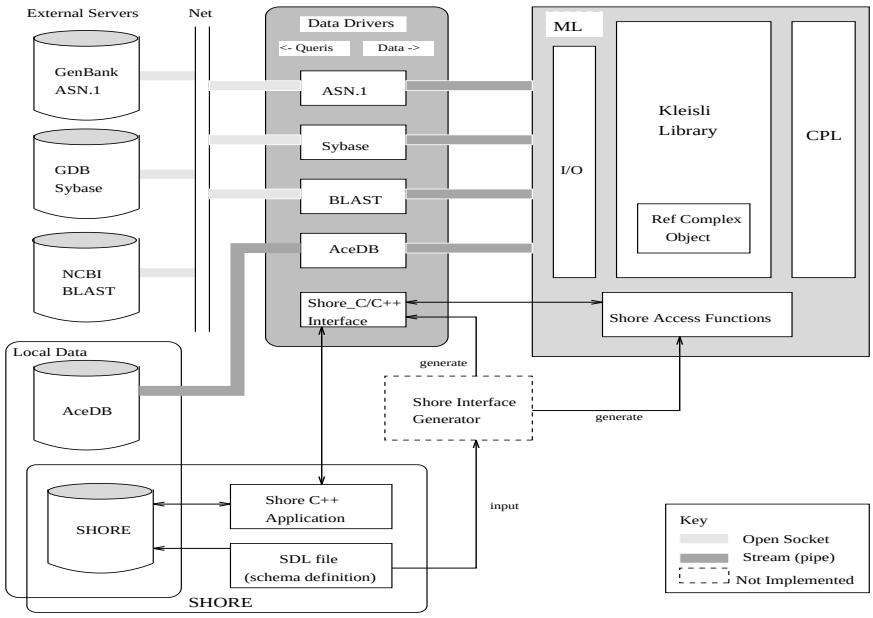

Figure 3: Architecture of the system

To connect to Shore [9], an object-oriented database under development at the University of Wisconsin, a different interface had to be developed since Shore does not support a high-level query language. The resulting interface is more tightly connected to the data source. Each object in a Shore database corresponds to a reference (a value of type class) in Kleisli, with associated functions to access its attributes and methods. These functions are represented in Figure 3 by the *Shore Access Functions*, and *Shore C/C++ interface* components.

A Shore application is composed of a schema definition, using the Shore Data Language (SDL), and the implementation of the methods in a language for which a binding is offered (currently only C++). To call a method or access an attribute from an object from Kleisli, the corresponding ML function in *Shore Access Function* is called, which in turn calls a C/C++ function in *Shore C/C++*; this finally calls the corresponding method implemented by a *Shore C++ application*. Note that the access functions in both components described above depend on the schema of the database, and must be generated for each Shore database to be integrated to Kleisli. This is performed by the *Shore Interface Generator* that takes as input the schema definition of a Shore database (an SDL file).

Kleisli has two interfaces: the application programming interface and the compiler interface. The application programming interface consists of ML modules implementing the data types supported in the model described in Section 4. Therefore, extending the system with a new data type requires basically the implementation of a new ML module. We extended Kleisli with a module for the reference complex object. The compiler interface supports the rapid constructions of query languages, as we have done for CPL in the present prototype, and contains modules which provide support for compiler/interpreter construction activities. In this part, the type inference mechanism were modified to handle recursive values that can be introduced in the presence of references.

The extension of CPL and Kleisli with the class type has been implemented, but the Shore Interface Generator has not been implemented yet. To test the feasibility of our approach a Shore School database used in the examples was connected to the system and is fully operational.

6 Conclusion

We have shown how CPL, a functional query language based on a complex value model, can be extended to query OODBs by the addition of a reference type and four operations: dereference, method invocation, identity test and class type cast. The extended language was shown to express a number of interesting queries, and has been implemented as a query interface to Shore databases, a bare-bones OODB developed at the University of Wisconsin. However, the question remains of how our model and language compares to the "industry standard" specified by ODMG [10].

The *Object Model* (OM) specified by ODMG using the *Object Definition Language* (ODL) is slightly richer than the model we presented in Section 3. Both models make a distinction between literals and objects, and both models support a rich set of collection types – sets, bags, lists, and arrays (see [16] for a discussion of arrays in CPL). However, ODL allows a richer set of base types than is currently supported in CPL (although such additions are not difficult to make), and it also supports null values for every literal type. In ODL, the type of values associated to objects is also defined by a class interface, composed of a set of attributes, relationships, and methods, or it can be a collection type. That is, collections can be objects.

OM allows multiple inheritance, however, little is said about how name conflicts between superclasses are resolved. In contrast, in our model conflicts must be explicitly resolved within a common superclass. Our model also requires the existence of class extents while they are optional within OM. If a class has an extent within OM, it can also define *keys*, which uniquely identify objects by the values they carry for some property or set of properties. It is also important to note that OM supports exceptions.

Turning to the languages, OQL and extended CPL are based on similar principles: both provide high-level primitives to deal with various collection types, and both are functional languages in which operators can be freely composed as long as the operands respect the type system. Many of the examples presented in Section 2 involving complex types are expressible in OQL by allowing a select-from-where clause in the select part.

In contrast, CPL uses comprehension syntax to manipulate sets, bags, and lists, and explicitly allows the programmer to convert between them. Moreover, since CPL includes primitive functions for comparing complex objects, it is possible to define precisely the result of converting list to sets or bags. For example, the expression $\{\!\|e_1 \mid \backslash x <\!\!- e_2 \|\!\}$ stands for the list $\{\!\|e_1[o_1/x], \ldots, e_1[o_n/x] \|\!\}$, where $o_1, \ldots o_n$ are the distinct elements in the set e_2 and $o_1 < \ldots < o_n$. As a consequence, the result of flattening a collection of collections of any type is well-defined and can be determined by looking at the type of the comprehension and the type of generators. On the other hand, in OQL flattening a list of sets, or a set of lists, always results in a set. Also, the result of a select-from-where clause can only be of type set or bag.

OQL supports operators to get elements from a collection. For example, it is possible to extract the element of a singleton, or the first and last elements of a list. CPL does *not* support such operations since they can

raise an exception. For instance, trying to extract the element of an empty set or a set with more than one element raises an exception. OQL also allows the creation of both complex values and objects, whereas in extended CPL only complex values can currently be created. Extended CPL also assumes that methods have no side effects. Although OQL allows the invocation of update methods, it does not include a formal specification of the semantics of these operations.

The languages also differ in how they support the movement of objects within the type hierarchy. In OQL, an object statically typed as of class C is allowed to go "up" in the hierarchy (i.e. typed as of a class C', C' superclass of C) as well as "down" (i.e. typed as of class C'', C'' subclass of C). The first conversion is always safe and it is done implicitly by OQL, however the second conversion can raise an exception. Therefore, extended CPL only allows movement "up" in the hierarchy, and requires that the user defines the conversion explicitly by using the operator as_C. This considerably simplifies type inference, and has been adopted in other systems such as Object ML [18].

Finally, in OQL any named object is an entry-point to a query, whereas in our approach only extents can be entry-points. OQL supports path expressions with any number of levels. For example, given an object p of class Person, one can obtain the age of p's mother as p.mother.age, where mother is also an object of class Person. If the value of p's mother is null, OQL raises an exception. To avoid this, a query may test explicitly if p.mother is not equal to null. Since CPL does not support nullable types, a class type is always defined as a variant, which always requires a test to extract its value.

From the discussion above, extended CPL can be thought of as a "safe" version of OQL since most of the differences between the two languages result from the avoidance of exceptions within extended CPL. The main drawback of our approach is the requirement for explicit subtyping using the as_C operation; on the other hand, the support within CPL for pattern matching is a very convenient mechanism to formulate queries and the language itself is simple and easy to use.

We have found extended CPL to provide an elegant query interface to OODBs with "programming" interfaces like Shore. As described in [6], it is possible to express in CPL most of the collection expressions found in other query languages: universal and existential quantification, membership testing, operations from the relational algebra (union, difference, product, selection, and projection), as well as the group-by construct of SQL. The extensions proposed in this paper additionally provide operations on object identity (equality and as_C), path expressions, method invocation, and late binding.

Implementing these extensions within the Kleisli system has also been remarkably easy. Since Kleisli is intended as a transformation and integration system for multiple heterogeneous data sources, it has been designed to facilitate the addition of external functions representing the query capabilities of data sources. Kleisli also supports an extensible query optimizer, which will also make it easy to add the optimizations sketched in Section 4.3. However, a complete treatment of optimizations for the extensions to CPL proposed in this paper remains an area of future research.

Acknowledgements

The authors thank Peter Buneman and Val Tannen for numerous fruitful discussions and suggestions during the development of this work.

References

[1] Abiteboul, S., Hull, R., Vianu, V., *Foundations of Databases*, Addison-Wesley Publishing Company, 1995

[2] Beeri, C., Kornatzky, Y., "Algebraic optimization of object oriented query languages", *Theorical Computer Science*, 116(1):59-94, August 1993

[3] Breazu-Tannen, V., Buneman, P, Wong, L. "Naturally embedded query languages". *LNCS 646: Proceedings of 4th International Conference on Database Theory*, Berlin, Germany, October 1992, J. Biskup and R. Hull, Eds., Springer-Verlag, pp. 140-154. Available as UPenn Techinical Report MS-CIS-92-47

[4] Breazu-Tannen, V., Buneman, P., Naqvi, S., "Structural recursion as a query language", *Proceedings of 3rd International Workshop on Database Programming Language*, Naphlion, Greece, August 1991, Morgan Kaufmann, pp. 9-19. Also available as UPenn Technical Report MS-CIS-92-17

[5] Buneman, P., Naqvi, S., Tannen, V., Wong, L., "Principles of Programming with Complex Objects and Collection Types", Theoretical Computer Science 149 (1995), pp 3-48.

[6] Buneman, P., Libkin, L., Suciu, D., Breazu-Tannen, V., Wong, L. "Comprehension Syntax", *SIGMOD Record*, 23(1):87-96, March 1994

[7] Buneman, P., Davidson S.B., Hart, K., Overton, C., Wong, L. "A Data Transformation System for Biological Data Sources". *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995

[8] Buneman, P., Ohori, A., "A Type System that Reconciles Classes and Extents", Proceedings of 3rd International Conference on Database Programming Languages, Nafplion, Greece, pp. 191-202, Morgan Kaufmann

[9] Carey, M., DeWitt, D., Naughton, J., Solomon, M., et al. "Shoring Up Persistent Applications". *Proceedings of the 1994 ACM SIGMOD Conference*, Minneapolis, MN, pp. 383-394, May 1994

[10] Cattell, R.G.G. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994

[11] Chawathe, S., Garcia, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J., "The TSIMMMIS Project: Integration of Heterogeneous Information Sources", *Proceedings of IPSJ Conference*, pp. 7-18, Tokyo, Japan, October 1994.

[12] Cluet, S., Delobel, C., "A General Framework for the Optimization of Object-Oriented Queries", *Proceedings of the 1992 ACM SIGMOD Conference*, 1992

[13] Deux, O., et al., "The Story of O2", *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1990

[14] Kosky, A. *Transforming Databases with Recursive Data Structures*, Ph.D. Thesis, University of Pennsylvania, 1996

[15] Levy, A., Srivastava, D., Kirk, T., "Data Model and Query Evaluation in Global Information Systems", Journal of Intelligent Information Systems, 1995. Special Issue on Networked Information Discovery and Retrieval, 5(2), September 1995

[16] Libkin, L, Machlin, R., Wong, L. "A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques", *Proceedings of the 1996 ACM SIGMOD Conference*

[17] Milner, R., Tofte, M., Harper, R. *The Definition of Standard ML*, MIT Press, 1990

[18] Reppy, J., Riecke, J., "Simple objects for Standard ML", *SIGPLAN'96: Conference on Programming Languages, Design, and Implementation (PLDI)*, 31(5), May 1996

[19] Trinder, P.W., Wadler, P. L. "Improving list comprehension database queries", *Proceedings of TENCON'89, Bombay, India*, November 1989, pp. 186-192

[20] Trinder, P.W. "Comprehensions, a query notation for DBPLs", *Proceedings of 3rd International Workshop on Database Programming Languages*, Nahplion, Greece, August 1991, pp. 49-62

[21] Wadler, P., "Comprehending monads", *Mathematical Structures in Computer Science 2*, 1992, 461-493

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

A $\mathcal{NRC}$: A Language based on the Set Monad

CPL is implemented based on the abstract language $\mathcal{NRC}$. Details of the language can be found in [5] and [22]. $\mathcal{NRC}$ is very similar to CPL, except that it does not use pattern matching and uses the restricted form of structural recursion $\bigcup\{e_1 \mid x \in e_2\}$ instead of the comprehension construct of CPL. The meaning