Polyglot: Extensions to Relational Databases for Sharable Types and Functions in a Multi-Language Environment

Linda G. DeMichiel   Donald D. Chamberlin   Bruce G. Lindsay
Rakesh Agrawal   Manish Arya

IBM Almaden Research Center
San Jose, California 95120

Abstract

Polyglot is an extensible relational database type system that supports inheritance, encapsulation, and dynamic method dispatch. It allows use from multiple application languages and permits objects to retain their behavior as they cross the boundary between database and application program. This paper describes the design of Polyglot, extensions to the SQL language to support the use of Polyglot types and methods, and the implementation of Polyglot in the Starburst relational database system.

1 Introduction

One of the most important trends in database management is the trend toward increasing the semantic content of stored data. Early database systems regarded data as a collection of stored values, whose interpretation was left to the applications that used them. More recent systems have regarded the semantic behavior of stored objects as an important resource that can be shared among applications. This trend is perhaps most evident in object-oriented database systems that allow users to define new data types that encapsulate both representation and behavior [4, 10, 13, 20]. In relational systems, the same trend can be seen in the design of extended relational databases that support user-defined data types and functions (for example, [18, 14, 7, 11, 9, 17]), and active databases (for example, [21, 19, 15]), in which assertions and triggers are used to maintain the global integrity of stored data. However, a broader objective can be envisioned: data types that manifest their behavior not only in database queries, but in one or more host languages as well. For example, suppose a user defines a `date` type, and a function that computes the difference between two dates. It is desirable that the difference function over dates be usable not only in database queries, but also to compare dates in application programs, including both dates that have been fetched from the database and transient dates that have never been entered into the database. Libraries of user-defined types and functions, shared by means of an extended relational database system, can provide a persistence mechanism for instances of user-defined types, a central repository for sharing commonly-used types and functions, and a common format for sharing data among programs written in diverse languages. In addition, they can be used to provide object-oriented features such as encapsulation, inheritance, and polymorphism for host languages in which these features would not otherwise be available.

2 Polyglot Objectives

The Polyglot project at the Almaden Research Center is investigating techniques for developing and supporting relational database libraries of shared types and functions in a multi-language environment. The main objectives and features of Polyglot are the following:

1. *Relational Database Support for User-Defined Abstract Data Types.* The Polyglot type system is designed for use with the relational data model. Currently, most relational systems support only a small number of predefined data types with a fixed repertoire of behavior, and users cannot specify new types or extend the behavior of existing types. Polyglot seeks to achieve upward compatibility with existing relational systems by extending their rudimentary type systems and thereby enriching the behavior of the data elements over which the relational model is defined. For this purpose, Polyglot provides a range of *metatypes* with differing functionalities and costs, including types that correspond to the representational types of existing relational systems, primitive abstract data types that do not support inheritance, and abstract data types that capture such ba-

sic features of object-oriented technology as data abstraction, multiple inheritance, subtyping, dynamic method dispatch, and encapsulation. All Polyglot objects are instances of types.

2. *Preservation of Type Behavior.* In some database systems supporting user-defined types, instances of a type have behavior only in the application domain, and data loses its behavior once it crosses the boundary from the application to the database domain. In other systems, instances of a type have a behavior only in the database domain (for example, `date` and `time` in DB2), and type-specific operations cannot be applied to them in the application domain. Polyglot is designed to allow application programs to create and manipulate transient instances of database types that manifest the same semantic behavior as instances of these types that are stored in the database, and to allow database objects to be fetched into application programs without losing their type-specific behavior. In addition, it allows instances of database types to be converted into representations specific to the application language, so that existing libraries written in the application language can be used and these instances manipulated in the application programming language.

3. *Separation of Type Definition from Instance Storage.* In Polyglot, the definition of a type does not imply a place to store its instances. It is possible to completely specify a Polyglot type schema without creating any instances or places to store those instances. Rather than restricting the instances of a type to type extents, Polyglot allows instances of a given type to be created or stored in different databases, in temporary variables, and in different persistent collections (e.g., tables). The persistence of an instance depends on where it is stored, not on its type.

4. *Support for Generic Functions and Multi-Methods.* Operations on the instances of types are defined by *generic functions*, where a generic function corresponds to a set of *methods* and the methods define the type-specific behavior of the generic function. A method is a function instance that is defined for a set of arguments of particular types and that can be executed for any objects that are instances of those types or their subtypes. Polyglot's function system is based on the *multi-method* paradigm, in which the selection of the method to be executed at run time depends on the types of all the arguments to the invocation, rather than the type of only a single argument, as in languages based on the message-passing paradigm (e.g., C++[6], Smalltalk[8], and Eiffel[16]). While multi-methods have been used in CLOS[2] and CommonLoops[3], the novel aspect of the Polyglot method system is that all method invocations are statically type-checked [1].

5. *Integration with Multiple Languages.* While most object-oriented systems require applications to be written in the same language as the one used to define types to the database, Polyglot is designed to allow use from multiple application programming languages. Polyglot does not attempt to supplant the type systems of those languages. Rather it is designed both to allow type libraries written in the application language to be exploited on data that has been obtained from the database, and to allow Polyglot methods to be applied to data that has been created in the application language domain or retrieved into the application language domain.

6. *Separation of Encapsulation of Type Behavior from Type Specification.* Persistent type systems must support not only the definition of new types, but also the controlled evolution of their behavior. The encapsulation mechanisms based on lexical scope that are used in programming languages are inadequate to control the ability to specify and modify representation-sensitive behavior of persistent types in database systems. Polyglot uses database authorization facilities to control which users can specify access to the representation or invoke the "private" methods of a type.

7. *Extensions to the SQL Language.* Polyglot extends the SQL language to support user-defined types and functions, and allows user-defined types and functions to be used both in SQL statements and in host-language application programs, while minimizing the necessary extensions to standard SQL.

2.1 Organization of this Paper

In this paper, we describe how Polyglot addresses these several goals. In Section 3 we describe the families of types provided as part of the base Polyglot system, including mechanisms that allow these types to be interfaced to the host-language type system. Section 4 discusses the Polyglot function system and its use of the multi-method paradigm. Section 5 describes Polyglot's approach to encapsulation using database authorization facilities. Section 6 describes extensions to SQL to support the use of Polyglot types and methods. Lastly, in Section 7 we describe our initial implementation of Polyglot as part of the Starburst relational database prototype [12].

3 Polyglot Types

In Polyglot, types are categorized into *metatypes* according to their semantics and functionality. The metatype determines the physical representation of instances of the type and the operations that are automatically defined on the type and its instances. There are currently five predefined metatypes in Polyglot: *base-type*, *tuple-type*, *rename-type*, *array-type*, and *language-type*. The Polyglot metatype system is designed to allow the addition of new metatypes.

In the discussion that follows, we use the following notational conventions. We use upper-case letters in an italic font to denote types and the corresponding lower-case letters to denote their instances. Thus, we will write a to denote an instance of type A. We use upper-case letters in a typewriter font to denote the names of types and lower-case letters in a typewriter font to indicate variables. We denote methods by *method signatures*, $\texttt{m(}T^1, T^2 \ldots, T^n\texttt{)} \rightarrow R$, where `m` is the name of the method, T^i is the type of the i^{th} formal argument, and R is the type of the result. Finally, in our figures, we draw an arrow from subtype to supertype to denote the subtype–supertype relationships among types.

3.1 Base Types

Base types are the built-in types of the underlying database system—that is, they correspond to the representational types that are typically used for fields in conventional relational systems, such as *integer*, *character string*, *byte string*, *boolean*, and *float*. The set of base types—unlike the sets of tuple types, rename types, array types, and language types—is not extensible from within Polyglot.

Operations on instances of base types include user-defined operations as well as system-supplied operations such as arithmetic operations and comparison operations. User-defined operations are defined on base types by writing methods that take base types as arguments. It is transparent to the client of a base type as to whether an operation is system-supplied or user-defined.

Polyglot does not support inheritance from base types for reasons of performance and compatibility with existing data. Allowing inheritance from base types would entail dynamic dispatch for methods defined exclusively on base types, and would require the use of type tags on instances of base types, which current relational database systems do not support.

3.2 Tuple Types

Tuple types are structured, abstract data types. A tuple type corresponds to a heterogeneous sequence of named fields, whose values may themselves be of any database type. Tuple types are encapsulated: the fields of an instance may be accessed only by methods defined on the type, and the use of these methods can be restricted. To the client of a tuple type, it is transparent as to whether such a method accesses an actual field of the tuple type instance or a virtual (computed) field. Tuple types support inheritance: a tuple type may be defined to inherit from one or more other tuple types (its *supertypes*).

When a tuple type is defined, a set of methods is automatically defined to provide primitive operations on the type. The automatically-defined methods created for a new tuple type T with fields $\texttt{f}_1, \ldots, \texttt{f}_n$ of types $F_1, \ldots, F_n$ respectively, include the following:

- A *constructor method*, $\texttt{new\_T()} \rightarrow T$, which constructs and returns a new instance of type T, all of whose fields have the value *null*.

- A set of *accessor methods*, $\texttt{f}_1\texttt{(}T\texttt{)} \rightarrow F_1, \ldots, \texttt{f}_n\texttt{(}T\texttt{)} \rightarrow F_n$, which retrieve the values of fields $\texttt{f}_1$, …, $\texttt{f}_n$ respectively.

- A set of *mutator methods*, $\texttt{set\_f}_1\texttt{(}T, F_1\texttt{)} \rightarrow T, \ldots, \texttt{set\_f}_n\texttt{(}T, F_n\texttt{)} \rightarrow T$, which set the values of fields $\texttt{f}_1$, …, $\texttt{f}_n$ respectively and return the modified instance. The new field values are copied into the tuple instance.

- A set of *downcasting functions* for T and each supertype S of T, $\texttt{is\_T(}S\texttt{)} \rightarrow T$, which test whether an instance of type T or any of its supertypes is of type T. The function `is_T` is the identity function when called with an instance of T or a subtype of T; when called with an instance of a proper supertype of T, it returns a null value of type T. These downcasting functions are designed to permit methods defined on type T to be safely applied to the elements of collections containing instances of both T and its supertypes.

- A *null predicate*, $\texttt{isnull(}T\texttt{)} \rightarrow Boolean$, which tests whether an instance is null or non-null.

- A *copier method*, $\texttt{copy(}T\texttt{)} \rightarrow T$, which returns a new copy of its argument.

A tuple type inherits structure and behavior from one or more other tuple types by designating those types as its supertypes in the type definition. The inheritance relation is transitive: a type inherits from its indirect as well as its direct supertypes. Structure is inherited in the sense that instances of the inheriting type have all the fields that were defined for instances of its supertypes in addition to fields that were defined for the inheriting type itself. Supertype operations are inherited in the sense that any functions that are defined for a supertype are applicable to instances of its subtypes as well. If two or more types from which a given tuple type inherits have a common supertype, the fields of that common supertype are inherited only once. If two or more types from which a given tuple type inherits have fields of the same name, all such fields are inherited by their common subtypes, and precedence of access to those fields is determined by the ordering of supertypes as specified in the type definition.

Consider the case where tuple type D multiply inherits from tuple types B and C, each of which inherits from tuple type A. Let us assume that the definition of type A specifies a field `v` of type V; the definition of type B specifies fields `w` of type W and `x` of type X_1; the definition of type C specifies fields `x` of type X_2 and `y` of type Y; and the definition of type D specifies a field `z` of type Z. Instances of tuple type D will have six fields: one corresponding to field `z` as defined by tuple type D itself; fields `w` and `x` from the definition of B, fields `x` and `y` from the definition of C, and field `v` from the definition of A. Note that the two `x` fields need not be of the same type. Because the fields of a tuple type are accessible only by means of accessor methods, from the standpoint of the structure of the instances of such a type, the names of such fields are incidental.

Because operations that are defined for a supertype are applicable to instances of its subtypes as well, there will be two `x` accessor methods and two `set_x` mutator methods applicable to instances of type D—those defined by type B and those defined by type C. These methods will access the fields specified by the definitions of B and C respectively. The order of type names in the inheritance specification is significant—it defines an inheritance precedence for the purpose of resolving method dispatch. Thus, in the above example, it will be the `x`-method defined for type B that will be executed when `x` is invoked on an instance of D, rather than the `x`-method defined for type C. Polyglot, however, allows methods to be directly specified, so that such "shadowed" methods are always available, as described in [5]. In Section 4, we will discuss the issue of method dispatch further.

3.3 Rename Types

The purpose of the rename metatype is to provide a primitive encapsulation mechanism that allows rename type instances to be treated as opaque entities. A rename type is obtained by encapsulating another database type. When a rename type is derived from a source type, that source type is made opaque but is otherwise not modified. Thus, at a representational level, instances of the source type and the rename type may be indistinguishable. However, because a rename type is an encapsulated type, only operations that are specifically defined for the rename type can be directly applied to its instances.

The automatically-defined methods created for a new rename type R of source type S include:

- An *upcasting* or instance creation method, `R(`S`)` $\rightarrow R$. Given an instance of the source type S, this operation returns an instance of the rename type R.
- A *downcasting* or value-access method, `S(`R`)` $\rightarrow S$. This operation allows an instance of the rename type to be accessed as an instance of the source type. It is expected that the value access operation will be protected, in which case the internal state of instances of the rename type will be encapsulated.
- A *null predicate*, `isnull(`R`)` $\rightarrow$ *Boolean*, which tests whether an instance is null or non-null.
- A *copier method*, `copy(`R`)` $\rightarrow R$, which returns a new copy of its argument.

Because instances of the source type and the rename type share a common representation, the upcasting and downcasting functions can be implemented with no runtime cost. Casting between a rename type and its source type, however, is never automatic. Thus, for example, if the rename type *distance* is created from the source type *integer*, then to add two distances, say `d1` and `d2`, using the integer addition operator `+`, it is necessary to write `distance(integer(d1) + integer(d2))`. It is expected that programming environment tools will provide convenient mechanisms and syntax for defining operations for rename types that are derived from operations on their source types.

Rename types are intended to provide an inexpensive, but functionally-limited abstract data type facility. Rename types provide name-equivalence type checking without supporting inheritance, and hence can be implemented without requiring dynamic dispatch or the use of type tags.

3.4 Array Types

The array metatype constructs a new type as a homogeneous, ordered, unbounded collection of elements of an existing type, accessed by an integer index. The element type of an array type can be any existing database type—hence array types can be nested to form arrays of multiple dimension. An array can be sparse (i.e., some of its interior elements can be null).

The automatically-defined functions for an array type A whose elements are of type E include:

- A *constructor method*, `new_A()`$\rightarrow A$, which constructs and returns a new, uninitialized array of type A.

- An *accessor method*, `get_element(`A`, `*integer*`)` $\rightarrow E$, which returns the indicated array element. A subscript notation for array accessor functions is also supported: the expression `A[`*integer*`]` is equivalent to the expression `get_element(`A`, `*integer*`)`.

- Two *mutator methods*, `set_element(`A`, `*integer*`, `E`)` $\rightarrow A$, which sets the indicated array element to the given value and returns the modified array, and `append(`A`, `E`)` $\rightarrow A$, which appends the given element immediately following the highest-index non-null element of the array and returns the modified array. If the array has no non-null elements, `append` causes the new element to be inserted at index zero.

- Methods that return the indices of non-null elements: `low_index(`A`)` $\rightarrow$ *integer*; `high_index(`A`)` $\rightarrow$ *integer*; and `next_index(`A`, `*integer*`)` $\rightarrow$ *integer*.

- A *null predicate*, `isnull(`A`)` $\rightarrow$ *Boolean*, which tests whether the array is null or non-null.

- A *copier method*, `copy(`A`)` $\rightarrow A$, which returns a new copy of the array.

3.5 Language Types

The purpose of the language metatype is to provide an interface between the database and programming language type systems. The language metatype specifies a host-language representation into which a database type can be transformed if necessary. Each language type is the representation for a specific database type in a specific host language. It is not necessary, however, for a database type to have a corresponding language type. Transforming an instance of a database type into the corresponding language type enables it to be operated on by existing host-language library functions that depend on a specific representation. If a Polyglot function is invoked on an instance of a language type, that instance will be transformed into its database representation before being passed as an argument to the function.

The only Polyglot operations that can be directly performed on instances of language types are the two conversion operations: `L2D` (language-to-database) and `D2L` (database-to-language). The `L2D` operation converts an instance of a language type to an instance of the corresponding database type. The `D2L` operation converts an instance of a database type to an instance of a specific language type. These conversion operations cannot be explicitly invoked. Instead, assignment to—or input from—a host variable of language type automatically invokes the appropriate conversion operation between the corresponding language type and database type.

The user who defines a language type for a new database type must provide the `L2D` and `D2L` functions that transform instances between the database and host-language representations. The `L2D` and `D2L` functions are written in the host language of the language type (this is necessary in order to interpret and initialize the internal structure of the language type instance). These functions are similar to the "input" and "output" functions required of user-defined data types in POSTGRES[18], except that they permit conversion of database types to and from structured language types rather than simply character strings. For each base type, Polyglot provides a corresponding language type in each supported host language and functions to convert between the database type and the language type.

In order to define a language type corresponding to a given Polyglot type, a host language must have some means of representing an instance of that type in its own type system. For example, a host language might represent the language form of a Polyglot tuple type by using a structured type of its own (e.g. a `struct` in C) or by some other means such as an array of pointers to more primitive types. To define a language type for a Polyglot type that has no predefined size, such as an array-type, a host language must either allocate memory dynamically or manage a sufficiently large buffer. It is important to note that a host language that lacks the ability to define a language type for a given Polyglot type can still create and manipulate instances of that Polyglot type. The representation of such an instance is managed by Polyglot and exists in the host program only in the form of a token.

4 Methods

In Polyglot, operations on the instances of types are defined by *generic functions*, where a generic function corresponds to a set of *methods* and the methods define the type-specific behavior of the generic function. Since Polyglot supports subtyping and inheritance, it is possible for a variable or a function argument to be bound at run time to an instance of any subtype of its static type. Function dispatch in Polyglot is therefore dynamic: the method selected for execution depends on the run-time types of the actual arguments to the function invocation, rather than on their static types.

All Polyglot methods are *multi-methods*, that is, the method to be dispatched is selected on the basis of the run-time types of *all* its arguments. Multi-methods are a powerful tool for modeling multiary operations. Since multi-method dispatch is based on the types of all arguments, and not just a single argument, multi-methods allow function behavior to be defined for a set of types. This is in contrast to many object-oriented languages such as C++ [6], Smalltalk [8], and Eiffel [16], which, while they allow multiary operations, are "selfish" in that a method provides type-specific behavior for only a single type. In such languages, there is a distinguished argument whose type determines which method is executed when a generic function is called. The remaining arguments provide values for the method that is selected, but they play no role in method selection itself. Thus, for example, while it is possible in such languages to write a generic function `area(shape)` that is dynamically dispatched based on the actual type of `shape` supplied, one cannot write a generic `display` function of the form `display(device, shape)` that is dynamically dispatched based on actual types of both `device` and `shape`.

4.1 Function Resolution

In principle, all function dispatch is Polyglot is dynamic—that is, the method that is chosen for execution depends on the types of the actual arguments to the function invocation, and is the *most specific applicable method* for those arguments. The problem of function resolution involves finding which of a set of methods are *applicable* to a given invocation, and, of these, which is the *most specific applicable method* for the given arguments. The *applicable* methods for a given function invocation are those methods that the user is authorized to invoke, and whose formal arguments are of the same types as, or supertypes of, the arguments of the invocation. Of these, the *most specific applicable method* is determined by considering each argument position in turn, from left to right. The method is selected whose first formal argument type is most specific according to the inheritance hierarchy; if a tie exists, the second formal argument is considered, and so on.

Thus, to continue with our previous example, a type definer might write methods `display`(*device, shape*), `display`(*screen, shape*), and `display`(*device, polygon*). If *screen* is a subtype of *device*, and *polygon* is a subtype of *shape*, then all three `display` methods will be applicable to arguments that are instances of the types *vt100* and *square* respectively, if *vt100* is a subtype of *screen* and *square* is a subtype of *polygon*. In this case, Polyglot multi-method dispatching will cause the method `display`(*screen, shape*) to be invoked.

4.2 Static Type Checking

While multi-methods have been used in CLOS [2] and CommonLoops [3], the novel aspect of the Polyglot function system is that all method invocations are type-checked at compile time. It is guaranteed for a type-checked Polyglot program that an object will never be assigned to a variable of incompatible type, and that the situation will never arise at run time in which there is no method that is applicable for a given set of actual arguments.

Static type checking in the absence of subtype polymorphism requires that we ensure at compile time that every operation receives the proper number of arguments, that these arguments are instances of the proper data types, and that the result type matches the expected result type. To allow for subtype polymorphism, this constraint is relaxed in the following way. Since a variable can be bound at run time to an instance of its static type or any subtype of its static type, to guarantee type safety we need only determine that the run-time type of each actual argument of a function invocation will be the same type as or a subtype of the static type of its corresponding formal argument, and that the result type of the function dispatched at run-time will be compatible with its invocation context. We also need to ensure that at least one method will be applicable at run time.

The above condition is satisfied by requiring that all generic functions be consistent in the following sense. We say that two methods $m_i(T_i^1, T_i^2, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n) \rightarrow R_j$ of a generic function M are *mutually consistent* if whenever they are both applicable for arguments of types $T^1, \ldots, T^n$, and m_i would be selected rather than m_j, then R_i is the same type as or a subtype of R_j. A generic function is *consistent* if all of its methods are mutually consistent. Note that this consistency requirement is more general than that of C++, which requires that all methods of a generic (*virtual*) function have the same result type [6].

For example, consider the types A, B, C, D, X, Y, and Z, where B is a subtype of A, D is a subtype of C, Z is a subtype of Y, and Y is a subtype of X. Assume that the generic function m consists of the following four methods:

$$\begin{array}{ll@{\qquad}c@{\qquad}c@{\qquad}c}
 & & & & \mathrm{X} \\
m_1(A,C) & \rightarrow X & \mathrm{A} & \mathrm{C} & \uparrow \\
m_2(B,C) & \rightarrow Y & \uparrow & \uparrow & \mathrm{Y} \\
m_3(A,D) & \rightarrow Y & \mathrm{B} & \mathrm{D} & \uparrow \\
m_4(B,D) & \rightarrow Z & & & \mathrm{Z}
\end{array}$$

The methods of m are pairwise mutually consistent. Thus, a call to m such as $x \leftarrow m(a, c)$, where x, a, and c are of static types X, A, and C respectively, is type safe, since no matter which subtypes of A and C occur as arguments at run-time, the method that is selected for execution will return a value that is a subtype of X. Furthermore, it is guaranteed that there will be at least one method for m that is applicable at run-time.

Given the requirement of mutual consistency of method definitions, the problem of the type checking of method invocations reduces to the following: determining that the most specific applicable method for the static argument types (determined as described in Section 4.1 above) returns a result type that is compatible with the context in which it occurs. Algorithms for the static type-checking and run-time dispatch of multimethods are described in detail in [1].

5 Encapsulation

In a "selfish" language such as C++, functions are strongly associated with types, and the internal representation of a type is accessible only to the functions associated with that type. This approach provides a desirable form of encapsulation, in which clients of a type can interact with the type only through those functions that are declared to be part of its external interface.

Rather than associating functions with types, Polyglot allows the definer of a type to protect its internal representation by means of an authorization subsystem, in which the *invocation privilege* on various functions can be granted to individual users or to the public.

For example, Polyglot can emulate the encapsulation facilities of a selfish system by creating each type, and the functions associated with that type, under a separate userid. The creator of the type and its functions can then grant the invocation privilege for those functions that comprise the external interface of the type to the public or to selected clients.

The Polyglot authorization subsystem is also sufficiently flexible to implement other useful encapsulation policies. For example, a user may wish to define a cluster of related types and functions that operate on combinations of those types. Thus, a geometry library might include types such as *Line* and *Polygon*, and functions such as `intersect`(*Line, Polygon*) which do not belong strictly to a single type. Using the authorization subsystem, a designer can encapsulate the geometry library as a whole, allowing its functions to have access to the representations of both lines and polygons, while presenting only an external interface to clients of the library.

6 Extensions to SQL

While many of the ideas being explored by the Polyglot project are platform-independent, the initial Polyglot implementation will be embedded in the Starburst relational database prototype [12], which supports an extended version of the SQL language. The principal Polyglot extensions to standard SQL include a `create type` statement, a `create function` statement, and a `set` statement that evaluates Polyglot expressions over transient data.

6.1 Create Type Statement

The `create type` statement is provided for defining new abstract data types of the extensible metatypes (tuple, rename, array, and language). The general syntax of the `create type` statement is the following:

```
create metatype type type-name attributes;
```

The attributes to the `create type` statement are specific to the metatype. The syntax for the four initial extensible metatypes is given below:

```
create rename type rename-type source source-type;

create tuple type tuple-type
       fields (field1 type1, field2 type2, ... ),
       inherits (supertype1, supertype2, ... );

create array type array-type of element-type;

create language type language-type
       for database-type,
       language host-language-name,
       declaration template,
       L2D function-name,
       D2L function-name;
```

The `declaration` attribute of the language type is a template for the host-language declaration that will be generated by the Polyglot precompiler for each instance of the language type. The `L2D` and `D2L` attributes name the user-supplied functions that convert an instance between the language type and its corresponding database type.

6.2 Create Function Statement

The syntax of the `create function` statement is the following:

```
create function function-name
     (arg-type-1, arg-type-2, ... )
       returns result-type,
       language implementation-language,
       file source-file-name,
       options;
```

The `create function` statement specifies the name of the function, the types of all its arguments, the language in which it is implemented, and the file in which its source code can be found by the Polyglot precompiler (alternatively, for short functions, the source code can be contained directly in the `create function` statement). The initial Polyglot implementation supports two function implementation languages, C++ and SQL. Functions written in C++ can contain SQL statements, and are precompiled in the same way as an application program. The object code of a Polyglot function is dynamically loaded and dispatched when needed.

The options that are accepted by the `create function` statement include the following:

- `nonulls`, meaning that the body of the function will never be executed if any of the arguments is null; instead, a null result will be returned.
- `protected`, meaning that use of the function is initially restricted to the function creator.
- `nondeterministic`, meaning that the function may not always return the same result for a given set of arguments. Note that nondeterminism limits the use of optimization techniques such as combining common subexpressions.
- `replace`, meaning that this function definition is to replace the definition of any previously-registered function of the same name and signature, preserving any existing authorizations.
- command-line options to be used in compiling and linking the function.

6.3 Set and Update Statements

Polyglot supports a new SQL statement called the `set` statement. The purpose of the `set` statement is to evaluate one or more expressions, optionally assigning copies of the resulting values to one or more target variables. The syntax of the `set` statement is the following:

`set [`*target* `:= ]` *expression* `[ , ... ];`

where the brackets indicate optional elements.

The `set` statement provides a means for an application program to invoke type-specific functions on transient and retrieved instances of database or language types. The expressions in a `set` statement may include Polyglot functions and variables declared with Polyglot types. To be consistent with standard SQL, the `set` statement prefixes the names of program variables with colons to distinguish them from the names of database objects such as tables and columns. The following examples illustrate the use of the `set` statement:

```
set :polygon3 :=
         intersect(:polygon1, :polygon2);

set :x := rating(:emp1),
    :y := compensation(:emp1);
```

In addition to altering the state of the target variables, a `set` statement can alter the state of a variable that is used as an argument to a mutator function. In the following example, `germany` is a variable whose value is a tuple of type *country*, and `set_capital` is a mutator function defined on the type *country*.

```
set set_capital(:germany, 'Berlin');
```

Polyglot also supports a "dot"-notation for invoking unary functions: The expression `T.x` is equivalent to `x(T)`. Although dot-notation can be used to invoke any unary function, it is typically used to invoke an accessor function on a tuple instance. A target variable in a `set` statement may be qualified by dot-notation to denote a field of a tuple, or by subscript-notation to denote an element of an array. These notations, used with target-variables, are shorthand ways of invoking mutator functions on instances of tuple and array types.

For consistency with the `set` statement, Polyglot also extends the syntax of the SQL `update` statement to permit the use of dot-notation and subscript-notation on the left side of the `set`-clause, and to permit the use of `:=` as an assignment operator, as shown below:

```
update emp
set address.zipcode := '95120',
    hobbies[3] := 'Sailing'
where name = 'Don';
```

6.4 The Use of Mutator Functions in SQL Queries

Mutator functions invoked within SQL queries can lead to nondeterministic results. This is because SQL does not guarantee the order in which predicates will be evaluated, and may in some cases omit the evaluation of a predicate entirely (e.g., if the predicate occurs in an `or` clause with another predicate that evaluates to true.) For example, consider the following query, in which `smooth` is a mutator function that alters the state of its polygon argument:

```
select name, border
from countries
where area(border) > 100000
or perimeter(smooth(border)) > 5000;
```

The `border` polygons retrieved by this query may or may not be smoothed, and if they are, the smoothing operation may have been applied either before or after the evaluation of the predicate `area(border) > 100000`.

In order to guard against such nondeterminism, Polyglot allows mutator functions to be invoked only in the following contexts:

1. In a `set` statement.
2. In the `set`-clause of an `update` statement, but only to mutate a database column that appears on the left side of the `set`-clause.

3. In any SQL statement, but only to mutate a value produced by a copier function or a constructor function. For example, the nondeterministic query above can be made deterministic (and legal) in the following way:

```
select name, border
from countries
where area(border) > 100000
or perimeter(smooth(copy(border))) > 5000;
```

The expected implementation of these rules is by means of *mutability flags* that are attached to the representations of data instances. Operations such as `new` and `copy` that create mutable instances set the associated flags to `mutable`. The primitive Polyglot mutator functions for tuple types and array types test the mutability flags of their arguments before proceeding. Since any user-defined mutator function must ultimately invoke a primitive Polyglot mutator, this mechanism is sufficient to protect against nondeterministic queries caused by mutator functions.

7 Implementation in Starburst

The implementation of Polyglot in Starburst will employ a precompiler that is similar to the precompilers used by many existing relational database products. For example, if a C program needs to operate on a database, it can do so by means of SQL statements that are preceded by the prefix `exec sql`. The precompiler recognizes the `exec sql` prefix, strips out the SQL statements, compiles them and stores them in the form of optimized plans for data access, and replaces the SQL statements by C code to call the run-time database system and execute the stored plans. Similarly, statements that invoke the type-specific behavior of Polyglot instances, either transient or persistent, will be prefixed by `exec sql` so that they can be detected by a precompiler and replaced with host-language code that invokes Polyglot methods at run time. The syntax that follows the `exec sql` prefix will be uniform across the various programming languages that Polyglot supports, and therefore will not be tightly integrated with any particular programming language. In other words, the initial Polyglot implementation draws a distinction between a Polyglot variable and a host-language variable. Polyglot functions and operators can be applied to Polyglot variables, and host-language functions and operators can be applied to host-language variables, and values can be exchanged between Polyglot variables and host-language variables; however, Polyglot functions and host-language functions cannot be mixed in the same expression.

In order to be used with Polyglot, a host programming language must be capable of interfacing to a C program, since the Starburst run-time facilities are written in C. The actual representations of Polyglot objects are maintained by Polyglot and are not interpreted by the host program. When invoking a Polyglot function, the host program passes tokens to Polyglot identifying the objects that are to be passed to the function (for example, if the host language is C++, the tokens are `void` pointers). The host language must also be capable of passing to Polyglot the addresses where function results and return codes are to be delivered.

During execution of a SQL statement, instances of abstract types are fetched from the database into a buffer pool along with instances of built-in types. User-defined methods are dispatched against the objects in the buffer pool, but for protection reasons they are not allowed to update the buffer pool directly. All updates to persistent data must be made by SQL `insert`, `update`, and `delete` statements, which invoke database functions such as locking, logging, and triggering of integrity assertions.

Transient variables of abstract data types are represented in a memory segment that is shared by Polyglot and the host program. The precompiler replaces the declarations of these variables with host-language declarations of tokens as described above. The host program, after precompilation, uses the tokens as references to the abstract objects, but does not interpret the representations of the objects themselves. When the host program calls for values to be exchanged between a Polyglot type and a language type, the appropriate `L2D` or `D2L` functions are invoked.

Polyglot expressions are evaluated by a simple stack-based interpreter called XPR. For each expression in a SQL statement, the precompiler generates code to push the necessary object references onto the XPR stack and to invoke the interpreter. The result of the expression is left on the stack, from which it can be assigned to a host-language variable or used in query processing. In order to avoid unnecessary context switches, separate copies of the expression interpreter run in the application address space (for operating on transient data) and in the database address space (for operating on persistent data).

Once a program has been precompiled, it is dependent on the latest version of all the types and functions that it uses. If any of these types or functions should be dropped or altered, or if additional types or functions should be defined that are confusable (e.g., through inheritance) with the existing ones, the program is automatically marked for recompilation before its next use.

8 Summary

We have described a set of extensions to relational databases that support the definition of new abstract data types and functions. Important features of the Polyglot design include multi-method dispatching and the ability of objects to retain their behavior as they cross the boundary between database and application program. These features permit the definition of libraries of abstract data types and functions that can be shared among applications in multiple host languages.

References

[1] R. Agrawal, L.G. DeMichiel, and B.G. Lindsay. "Static Type Checking of Multi-Methods." *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), 1991.

[2] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification.* Document 88-002R, X3J13, 1988. Also published in *SIGPLAN Notices*, Vol. 23, special issue, Sept. 1988.

[3] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. "CommonLoops: Merging Lisp and Object-Oriented Programming." *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), 1986.

[4] R. Cattell. *Object Data Management.* Addison-Wesley, 1991.

[5] L.G. DeMichiel, D.D. Chamberlin, B.G. Lindsay, R. Agrawal, and M. Arya, "Polyglot: Extensions to Relational Databases for Sharable Types and Functions in a Multi-Language Environment," IBM Research Report, RJ 8888, 1992.

[6] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[7] G. Gardarin, J-P. Cheiney, G. Kiernan, D. Pastre, and H. Stora, "Managing Complex Objects in an Extensible Relational DBMS." *Proc. of Very Large Data Base Conference* (VLDB), 1989.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[9] R.H. Güting. "Gral: An Extensible Relational Database System for Geomatric Applications." *Proc. of Very Large Data Base Conference* (VLDB), 1989.

[10] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[11] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Sudkamp, G. Walch, and M. Wallrath. "Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions." *Proc. of Very Large Data Base Conference* (VLDB), 1988.

[12] G. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. "Extensions to Starburst: Objects, Types, Functions, and Rules." *Communications of the ACM*, 34:10, October 1991.

[13] D. Maier and S.B. Zdonik. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.

[14] F. Manola and U. Dayal. "PDM: An Object-Oriented Data Model." *Proc. International Workshop on Object-Oriented Database Systems*, 1986.

[15] D.R. McCarthy and U. Dayal. "The Architecture of an Active Database Management System." *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1989.

[16] B. Meyer. *Eiffel: The Language.* Technical Report TR-EI-17/RM, Interactive Software Engineering Inc., 1989.

[17] S.L. Osborn and T.E. Heaven. "The Design of a Relational Database System with Abstract Data Types for Domains." *ACM Transactions on Database Systems*, Vol.11, No.3, Sept. 1986.

[18] M. Stonebraker. "Inclusion of New Types in Relational Database Systems." *Proc. of Second International Conf. on Database Engineering*, 1986.

[19] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. "On Rules, Procedures, Caching and Views in Database Systems." *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1990.

[20] G. Vossen. "Bibliography on Object-Oriented Database Management." *SIGMOD Record*, Vol. 20, No. 1., Mar. 1991.

[21] J. Widom, R.J. Cochrane, and B.G. Lindsay. "Implementing Set-Oriented Production Rules as an Extension to Starburst." *Proc. of Very Large Data Base Conference* (VLDB), 1991.