

Type System of an Object-Oriented Database Programming Language*

Extended abstract

Y. Leontiev, M. Tamer Özsu, Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E1
{yuri,ozsu,duane}@cs.ualberta.ca

Abstract

In this paper we present the type system of the TIGUKAT database programming language. It is a highly parametric object-oriented type system that combines multiple dispatch with reflexivity, separation of interface and implementation, precise behavior typing, and union and intersection types. We demonstrate the inner workings of the type system by considering a concrete example of type specification in TIGUKAT. We also review type systems of several existing programming languages and conclude that the proposed type system has a unique combination of features particularly suited for object-oriented database programming.

*This is an expanded version of the extended abstract submitted to DBPL 97.

1 Introduction

In the past two decades, several new database application areas have emerged. These new areas include office automation systems, geographical information systems, CASE tools, medical systems, and CAD/CAM systems. These complicated applications demand new, more advanced data modeling capabilities as well as a new level of integration between databases and programming languages.

One of the new data models aimed towards these new applications is the object-oriented data model. Object-oriented database systems have recently received much attention and are still one of the prolific areas of research and development activity.

In order to provide the necessary level of integration between programming languages and databases, a concept of a *database programming language* was introduced. Two decades of research in this area have led to the concept of an *orthogonally persistent object system* [AM95]. Orthogonally persistent object systems aim to provide seamless integration between databases and programming languages thus eliminating the impedance mismatch problem.

Object-oriented database programming languages (OODBPLs) have a potential to combine the modeling and software construction power of the object-oriented paradigm, extensive and efficient data storage and retrieval techniques of modern database systems, and the efficiency and power of today's programming languages in a single uniform framework. In order to achieve this goal, an OODBPL has to satisfy not only the requirements of object-oriented database systems [ABD⁺92], but also those of database programming languages [AB87] and orthogonally persistent object systems [AM95].

One of the most important components of any object-oriented programming language is its type system. The flexibility and expressiveness of the type system has a significant impact on the language. The type system of an OODBPL has an even greater impact on its characteristics since it corresponds to the data model of its database component [AM95]. Another reason why the expressibility and flexibility of a type system is so important in the context of an OODBPL is the fact that one of the desirable features of such a language is strong type checking.

The requirements that a type system of an OODBPL should satisfy thus follow from the requirements that must be met by an OODBPL itself. For example, an OODBPL must have declarative querying capabilities and therefore its type system must be able to correctly type queries. In order to do that, the type system should be able to deal with parametric collection types as well as with union and intersection types [PSVS95].

The requirements that must be met by a type system of an OODBPL can be categorized as those related to the type system expressibility, uniformity, safety, and reflexiveness.

Expressibility requirements deal with the ability of the type system to adequately model abstract and real-world entities. An expressive OODBPL type system should support parametric and inclusion polymorphism, multiple inheritance, multiple dispatch, higher-order function types, mutable object types, union and intersection types, and data encapsulation. The latter can easily be achieved if the type system supports separation between interface and implementation.

Uniformity is needed to support orthogonal persistence. It also has a great impact on the type system reflexivity, affects the ease of use of the language, and simplifies code reuse. A uniform type system does not make a distinction between methods and attributes at the interface level, provides the same status for system- and user-defined types, does not distinguish between objects and values at the interface level, and does not have special provisions for persistent or transient types (orthogonal persistence).

Safety requires that a type system support both static and strong type checking. Static type checking has long been advocated as one of the techniques that greatly reduces development time by catching many errors automatically at compile time. Strong type checking ensures that polymorphism does not introduce run-time type errors. In a persistent environment, the importance of strong and static type checking increases since

programs are persistent and thus the errors in them are persistent as well. Of course, there are situations where dynamic type checking is needed; however, it is desirable that the type system be able to statically type check as much of the code as possible.

Reflexiveness of the type system allows a user to query and manipulate a database schema using the same mechanisms that are used to query and manipulate ordinary data. In a language with a reflexive type system, types, behaviors, functions, and classes are objects and thus have types themselves.

There are type systems that satisfy *some* of the above requirements. However, it is a challenge to develop a consistent and uniform type system that satisfies them all.

This paper presents the type system of the TIGUKAT database programming language currently under development at the University of Alberta. We believe this type system to be a major contribution to the area of object-oriented database programming languages as it combines impressive modeling power with static type checking, reflexivity, and type system features specifically designed for database applications, such as union and intersection types. We believe that the TIGUKAT type system satisfies all the requirements listed above.

The paper is organized as follows: Section 2 briefly describes the TIGUKAT object model. Section 3 presents the type system of the TIGUKAT language and an example of its usage along with a discussion of the issues related to it, such as type checking and dispatch. We proceed to briefly discuss the implementation type system of TIGUKAT in Section 4, its class system (Section 5), and its reflexive capabilities in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2 TIGUKAT object model

TIGUKAT [ÖPS⁺95]¹ is an object database management system (ODBMS) based on a uniform behavioral object model. Basic notions of the model are *objects*, *atomic values*, *types* (denoted by a prefix **T**), *implementation types* (prefix **IT**), *behaviors* (prefix **B**), *classes* (prefix **C**), *functions* (prefix **F**), and *implementation functions* (prefix **IF**). We will consider these notions in turn.

Object: Object is a primitive term. In TIGUKAT, everything is an object. This includes atomic values, types, behaviors, classes, and functions. Examples of objects are 1, 'c', and **Person**.

Atomic value: In TIGUKAT, atomic values are objects. Atomic values are immutable and can not be created or deleted. Examples of atomic values are booleans, characters, integers, real numbers etc.

Type: A type in TIGUKAT is an object that describes behaviors applicable to the objects of this type. We say that a behavior is *defined on* a type if it is applicable to objects of this type. Every object in TIGUKAT has a unique type. We say that an object *o* maps to type **T.t** and write $o \mapsto \mathbf{T.t}$. Types form a directed acyclic graph (DAG) with respect to a subtyping relationship \preceq . Every type in TIGUKAT is a subtype of the root type **T.Object** and a supertype of the base type **T.Null**, except for **T.Error**, that is the only subtype of **T.Null**. **T.Object** and **T.Error** are not \top and \perp as they can define behaviors; in fact, there are quite a few behaviors defined on **T.Object**. Subtyping in TIGUKAT implies substitutability (every object that maps to a type can be used everywhere an object that maps to its supertype can). Subtyping in TIGUKAT is defined explicitly and implies inheritance of behaviors. Since TIGUKAT types are objects, they map to subtypes of a designated (meta-)type **T.Type**.

Behavior: A behavior in TIGUKAT is an analog to a *message* in more traditional object-oriented models. However, in TIGUKAT, a behavior is not just a name, but an object that has many behaviors defined

¹The model described here is a significant extension of the one described in [ÖPS⁺95].

on it. A behavior application (written as $\circ.B_b$) is the only way to perform an action in TIGUKAT. Behavior application $\circ.B_b$ is *valid* iff a behavior B_b is defined on a type T_t such that $\circ \mapsto T_t$. Behaviors are objects that map to subtypes of a designated type T_{Behavior} .

Class: Classes in TIGUKAT are used to maintain extents of types and to automatically group objects of the same type. Object creation in TIGUKAT occurs only via classes (for example, to create an object of type T_{Person} , one would write $C_{\text{Person}}.B_{\text{New}}$). A type can have no more than one class; at the same time, several types can share a class. TIGUKAT classes define neither object structure (that is defined by implementation types) nor object interface (defined by types). They are extent managers, and that is their primary purpose.

Function: Functions in TIGUKAT are analogous to *methods* in other object models and are pieces of code that get executed when a behavior is applied. Functions should be *associated* with a particular behavior on a particular type in order to be invoked when the behavior is applied to an object of that type. TIGUKAT supports the concept of late-binding; thus, when a behavior is applied to an object, the function to be executed is chosen dynamically according to the receiver type and all argument types (multiple dispatch). Functions are objects that map to subtypes of T_{Function} . Function code is written in the TIGUKAT database programming language; lower-level implementation functions can be used when an optimization or low-level access is required.

Implementation type: An implementation type in TIGUKAT is an object that describes the structure of objects of this implementation type. Implementation types form a single-inheritance (tree) hierarchy independent of the type hierarchy. Thus, unrelated types can use the same implementation type and vice versa. For example, both T_{Weight} and T_{Year} can be implemented via IT_{Integer} . Implementation subtyping will be denoted \preceq_{IT} throughout this paper. A type can have many implementation types; likewise, an implementation type can be used to implement objects of many different types. Implementation types are directly used by classes, and every concrete class has at least one implementation type associated with it. Types refer to implementation types indirectly, via classes. Thus, abstract types do not refer to any implementation type. Implementation types map to $T_{\text{ImplementationType}}$.

Implementation function: Implementation functions are designed to provide implementation type specific, optimized non-TIGUKAT code. Implementation functions are chosen according to the implementation types of actual arguments, just like functions are chosen according to the types of arguments. Implementation functions should be *associated* with a particular function on a particular product of implementation types in order to be invoked when the behavior is applied to objects of those implementation types.

TIGUKAT uses *two-phase dispatch*. A behavior applied to a set of arguments uses their types to obtain a function (multiple dispatch of behaviors) which then uses implementation types of the arguments to obtain the (executable) implementation function (multiple dispatch of functions). If the latter fails, the TIGUKAT code for the function is used. Thus in the simplest case when the specialization on implementation types is absent, TIGUKAT dispatch is equivalent to a standard multiple dispatch.

This is an extension of the standard notion of dispatch. The reason for using two-phase dispatch is the complete separation between *interface* and *implementation* in TIGUKAT. Implementation features are represented by implementation types, and implementation functions deal with these features. On the other hand, external object interfaces are defined by types and behaviors. Functions deal with semantically different operations related to the behaviors. A function code is thus independent of any implementation details, including the internal structure of objects.

The TIGUKAT model allows dynamic type, class, behavior, and function creation, modification, and

deletion. The issues related to schema evolution in the TIGUKAT object model are discussed in [PÖ95a] and [PÖ95b].

Persistence in TIGUKAT is orthogonal (independent of type) [ÖPS⁺95]. There are also some additional considerations about TIGUKAT persistence related to the structure of the model (for example, the type and the class of a persistent object have to be persistent as well); however, these additional complications do not affect the type system issues discussed in this paper.

Having briefly described the TIGUKAT object model, we now proceed to the description of the TIGUKAT type system.

3 TIGUKAT type system

The TIGUKAT type system consists of TIGUKAT types, behaviors, functions, and rules related to type specification, subtyping, inheritance, and type checking. Implementation types and implementation functions form their own *implementation type system* which is discussed in Section 4.

The TIGUKAT type system has multiple inheritance and parametricity, and is designed to support multiple dispatch of behaviors. Thus, it is very flexible and is capable of supporting a wide range of current applications. The reflexivity and extensibility of the type system, which is a consequence of the reflexivity of the TIGUKAT object model, allows one to design type system extensions suitable for specific problem domains. Examples of such development include an extensible query optimizer for the TIGUKAT query language [ÖMS95] and specification of an extensible temporal model [GLÖS95].

The following example will be used throughout the rest of the paper to illustrate various features of the TIGUKAT type and class system.

Example 3.1 Let us assume that we need to define real and integer numbers, as well as points and colored points, with partial orderings on them. We also would like to abstract out the property of being ordered (partially or totally) so that we can reuse the code for reverse ordering in both numbers and points. We would also like the ordering of color points to take color into account assuming that colors are also partially ordered (for example according to their RGB values). However, the comparison between a point and a color point should *not* take color into account. We would also like to write an optimized version of code that deals with comparison of numbers using their internal representation². Another requirement is that the system should be able to statically mark any attempted comparison between a number and a point as a type error, while allowing cross-comparisons between numbers or points of different types. The Figure 1 provides the TIGUKAT definitions of types related to the problem domain of this example. Parametric types **T_PartiallyOrdered** and **T_Ordered** provide the necessary abstraction of the ordering primitives together with the reusable code for reverse ordering (behavior *B_GreaterOrEqual*). Numeric types **T_Real** and its subtype **T_Integer** are also defined together with the function *F_LessOrEqualR* that provides optimized code for numeric comparisons. Point types define the code to perform necessary comparisons between points.

3.1 Parametric types and behaviors

In this section we describe TIGUKAT parametric types and behaviors. Parametric polymorphism is important for type system expressibility [AM95]. The combination of inclusion polymorphism inherent in object-oriented systems and parametric polymorphism provides certain expressive features not available in systems that support only one kind of polymorphism.

²Of course, the optimized comparison of numbers is a part of the kernel TIGUKAT system. We use it here for exemplification only.

```

type T_PartiallyOrdered(novar X) {
  supertype T_Object ;
  behavior B_LessOrEqual { x:T_PartiallyOrdered(X) →T_Boolean }
  behavior B_GreaterOrEqual { x:T_PartiallyOrdered(X) →T_Boolean :
    function { body { return x.B_LessOrEqual(self) } } }
type T_Ordered(novar X) {
  supertype T_PartiallyOrdered(X) ;
  behavior B_Less { x:T_Ordered(X) →T_Boolean :
    function { body { return self.B_LessOrEqual(x) ∧ self.B_NotEqual(x) } } }
  behavior B_Greater { x:T_Ordered(X) →T_Boolean :
    function { body { return self.B_GreaterOrEqual(x) ∧ self.B_NotEqual(x) } } }
type T_Point {
  supertype T_Object ;
  supertype T_PartiallyOrdered(T_Point);
  behavior B_LessOrEqual { x:T_Point →T_Boolean : function F_LessOrEqualPoint { body {
    return self.B_X.B_LessOrEqual(x.B_X) ∧ self.B_Y.B_LessOrEqual(x.B_Y) } } }
  behavior B_X { ↔T_Real : function { stored } }
  behavior B_Y { ↔T_Real : function { stored } } ... }
type T_ColorPoint {
  supertype T_Point ;
  behavior B_LessOrEqual { x:T_ColorPoint →T_Boolean : function F_LessOrEqualColorPoint {
    body { return self.B_X.B_LessOrEqual(x.B_X) ∧ self.B_Y.B_LessOrEqual(x.B_Y) ∧
    self.B_Color.B_LessOrEqual(x.B_Color) } } }
  behavior B_Color { ↔T_Color : function { stored } } ... }
type T_Real {
  supertype T_Ordered(T_Number) ;
  class C_Real { itype IT_Real { supertype IT_Atomic; ... } ... }
  behavior B_LessOrEqual { T_Real →T_Boolean : function F_LessOrEqualR ; } ... }
type T_Integer {
  supertype T_Real;
  class C_Integer { itype IT_Integer { supertype IT_Atomic; ... } ... }
  behavior B_LessOrEqual { T_Integer →T_Boolean : function F_LessOrEqualR ; } ... }
function F_LessOrEqualR {
  IT_Real, IT_Real : ifunction { /* low-level code of the function that compares two real numbers */ },
  IT_Real, IT_Integer : ifunction { /* low-level code of the function that compares real to an integer */ },
  IT_Integer, IT_Real : ifunction { /* low-level code of the function that compares integer to a real */ },
  IT_Integer, IT_Integer : ifunction { /* low-level code of the function that compares two integers */ }

```

Figure 1: Definitions of types of the Example 3.1

The TIGUKAT type system has very powerful parametric type specification capabilities. It is possible to define parametric types with several parameters, describe subtyping relationships between different parametric types from the same family, and create new sub- and super-types of existing parametric types. Parametric types can also be defined and manipulated at run-time, just like any other objects in TIGUKAT.

In TIGUKAT, parametric types form their own subhierarchies integrated into the TIGUKAT type hierarchy. Parametric types of the same kind always have a common supertype which is defined automatically. The name of this supertype of all types of the form $\mathbf{T_P}\langle X \rangle$ is conventionally written as $\mathbf{T_P}$. For example, the definition of $\mathbf{T_Ordered}\langle X \rangle$ (Figure 1) implicitly defines the type $\mathbf{T_Ordered}$ such that $\mathbf{T_Ordered}\langle X \rangle \preceq \mathbf{T_Ordered}$ for all types X . Another object that is implicitly defined by the above definition is the behavior $B_Ordered$ (defined on $\mathbf{T_Type}$ to yield a result of type $\mathbf{T_Type}$). The semantics of this behavior is given by the following definition: $\mathbf{T_X.B_Ordered} \equiv \mathbf{T_Ordered}\langle \mathbf{T_X} \rangle$. This behavior can be used at run-time to generate types of the form $\mathbf{T_Ordered}\langle X \rangle$.

Another aspect of the TIGUKAT parametric type specification is the ability to specify subtyping relationships between different types from the same parametric family. This is done by placing *variance specifications* beside the formal parameters in type specifications. For example, both $\mathbf{T_Ordered}\langle X \rangle$ and $\mathbf{T_PartiallyOrdered}\langle X \rangle$ are specified as being *invariant* (keyword **invar**) on their first and only argument. Invariance means that no subtyping relationship exists between $\mathbf{T_Ordered}\langle X \rangle$ and $\mathbf{T_Ordered}\langle Y \rangle$. Other possibilities include covariance (**covar**) and contravariance (**contravar**). Covariance means that for all types X and Y , $X \preceq Y \Rightarrow \mathbf{T_P}\langle X \rangle \preceq \mathbf{T_P}\langle Y \rangle$. Contravariance inverts this relationship. Types of updatable objects tend to be invariant, types of read-only objects tend to be covariant, and types of write-only objects tend to be contravariant (such as $\mathbf{T_OutputStream}\langle \mathbf{contravar} X \rangle$). A parametric type can have several parameters, each with its own variance specification.

It is also possible to specify subtyping between two different families of parametric types. For example, $\mathbf{T_Ordered}\langle X \rangle \preceq \mathbf{T_PartiallyOrdered}\langle X \rangle$ for all types X (Figure 1).

Type parameters in TIGUKAT can be restricted to be subtypes of some type (constrained parametric polymorphism). This allows us to constrain parameters of a certain parametric type to be, for example, subtypes of $\mathbf{T_Ordered}\langle X \rangle$.

Parameters of parametric types do not have to be types themselves. Objects of any type X can be used as parameters provided that the parametric type they are used for has been specified to accept parameters of this type. Invariant parameters can be of any type; however, covariant and contravariant parameters have to be of some type X that is a subtype of $\mathbf{T_PartiallyOrdered}\langle Y \rangle$ for some Y . For example, integer numbers can be used as parameters with any variance specification. This allows us to easily model such objects as arrays of fixed length.

Parametric types can be subtyped in exactly the same manner as ordinary ones. For example, the type of strings $\mathbf{T_String}$ is defined as a subtype of $\mathbf{T_List}\langle \mathbf{T_Character} \rangle$. User-defined and system-defined parametric types have exactly the same system status.

However, not all parametric types are created equal. Most of the parametric types in TIGUKAT are so-called *light parametric types*. The types from these parametric families are used for type checking only and are identical from the point of view of TIGUKAT dispatch. They are also modeled as atomic objects in the system, so that the system does not need to keep track of them. For example, all types $\mathbf{T_Ordered}\langle X \rangle$ defined in Figure 1 are equivalent to $\mathbf{T_Ordered}$ (their common supertype) from the point of view of the dispatch mechanism. This places several restrictions on function association. However, we have discovered that for the vast majority of parametric types these restrictions are never violated; besides, it is always possible to provide different implementations for parametric types from the same family using implementation types, functions, and the implementation dispatch mechanism. This treatment of light parametric types allows the system to deal with infiniteness of the type hierarchy. For example, at any given moment there is (theoretically) an

infinite number of set types ($\mathbf{T_Set}\langle X \rangle$) in the system. This is a great advantage in database settings, where a new set type can be easily (and implicitly) produced as a result of a set-theoretic operation, such as union.

However, there exist parametric types that do require a full-blown system representation. These are called *heavy parametric types* and can be introduced in almost the same way as light ones. At any given moment, there is only a fixed number of these in the system, and each such type is represented by a real object (not atomic). Of the kernel parametric types, only class types (Section 5) and $\mathbf{T_Var}\langle X \rangle$ (Section 3.5) are heavy.

Not only types, but also behaviors can be parametric. A parametric behavior $B_B\langle X \rangle$ essentially accepts additional arguments described by X , but does so statically, so that the typing can be made more precise. For example, consider the behavior $B_{As}\langle X \rangle$ that can be used to dynamically check the type of an object. It is defined (on $\mathbf{T_Object}$) as $B_{As}\langle X \rangle: \mathbf{T_Object} \rightarrow X$ and has the following semantics: if the receiver's type is a subtype of X , this behavior is equivalent to B_{Self} (identity function); otherwise, the behavior returns object `typeMismatch` of type $\mathbf{T_Error}$. Thus, the result of the behavior is guaranteed to be of type X and can be used in subsequent computations as such. This behavior can be used to implement a `typecase` operator similar to the one found in [LML⁺94]. If not for parametricity, we would have to specify this behavior as $B_{As}: \mathbf{T_Object}, \mathbf{T_Type} \rightarrow \mathbf{T_Object}$ and could not guarantee the result type. Of course, as with every mechanism in TIGUKAT, there exists a dynamic version: for each parametric behavior $B_B\langle X: \mathbf{T_X} \rangle$ there exists the automatically created *generating* behavior $B_{B_B}: \mathbf{T_X} \rightarrow \mathbf{T_Behavior}$.

The combination of features described in this section makes TIGUKAT parametricity an extremely powerful type specification mechanism. Another powerful mechanism, rooted in multiple dispatch, will be discussed in the next subsection.

3.2 Covariant specification and multiple dispatch

It is well known that multiple dispatch (dispatch that takes into account all arguments and not just the receiver) greatly enhances the expressiveness of the type specification, allowing for such notions as binary methods to be expressed naturally ([FE96], [ADL91], [Ghe91], [MHH91], [BG95], [CL94], [BC95], [Cas95], [CGL95], and [BCC⁺96]). Example 3.1 illustrates this power by providing “covariant” specification of comparison behaviors. In this section we examine the TIGUKAT type specification and multiple dispatch mechanisms and show how they combine to produce natural specification of binary methods.

First, we look into the specification of the behavior $B_{LessOrEqual}$. It is defined on the type $\mathbf{T_PartiallyOrdered}\langle X \rangle$ with an argument of type $\mathbf{T_PartiallyOrdered}\langle X \rangle$ and return type $\mathbf{T_Boolean}$ (Figure 1). We use the *product notation* (a notation in which the types of all arguments, including the receiver, are written on the left of the arrow, while the result type is written on the right) in our discussion of behavior specification and multiple dispatch. In this notation, the abovementioned behavior specification is written as $B_{LessOrEqual}: \mathbf{T_PartiallyOrdered}\langle X \rangle, \mathbf{T_PartiallyOrdered}\langle X \rangle \rightarrow \mathbf{T_Boolean}$. The function $F_{LessOrEqualR}$ is associated with this behavior by $\mathbf{T_Real}$ with argument of type $\mathbf{T_Real}$ (Figure 1); we will write this as $B_{LessOrEqual}: \mathbf{T_Real}, \mathbf{T_Real} \rightarrow \mathbf{T_Boolean} : F_{LessOrEqualR}$. Analogously, associations made by the types $\mathbf{T_Point}$ and $\mathbf{T_ColorPoint}$ (Figure 1) are written as $B_{LessOrEqual}: \mathbf{T_Point}, \mathbf{T_Point} \rightarrow \mathbf{T_Boolean} : F_{LessOrEqualPoint}$ and as $B_{LessOrEqual}: \mathbf{T_ColorPoint}, \mathbf{T_ColorPoint} \rightarrow \mathbf{T_Boolean} : F_{LessOrEqualColorPoint}$, respectively.

During type checking, the system tries to find a path in the subtype hierarchy between the known types of arguments and a type that unifies with the specification (in this case, any type of the form $\mathbf{T_PartiallyOrdered}\langle X \rangle \otimes \mathbf{T_PartiallyOrdered}\langle X \rangle$ where \otimes denotes a product of types). If such a path is found, the behavior application is said to be type-correct. Otherwise, it is rejected.

Consider type checking of the expression `aPoint.B_LessOrEqual(5.0)`, where `aPoint` is of type `T_Point` and `5.0` is of type `T_Real`. `T_Real` is known to be a subtype of `T_Ordered(T_Real)` and, therefore, a subtype of `T_PartiallyOrdered(T_Real)`. The type `T_Point`, on the other hand, is known to be a subtype of `T_PartiallyOrdered(T_Point)`. However, there is no way to unify `T_PartiallyOrdered(T_Point) ⊗ T_PartiallyOrdered(T_Real)` with `T_PartiallyOrdered(X) ⊗ T_PartiallyOrdered(X)` since the latter requires that both arguments be the same. Since `T_PartiallyOrdered` is nonvariant and therefore no other paths are possible, the type checking fails. If `T_PartiallyOrdered` was declared covariant, we would be able to find a unifier (`T_PartiallyOrdered(T_Object) ⊗ T_PartiallyOrdered(T_Object)`), since both `T_Real` and `T_Point` are subtypes of `T_Object`) and the type checking would succeed.

During dispatch, the system considers all function associations and tries to unify the actual arguments' product type with their arguments. If the unification succeeds, the distance from the actual arguments' product type to the unifier type is calculated. The function association that has the minimal distance is chosen, and its function is executed. The system statically checks that such an association always exists for any actual arguments that might occur and that it is always unique. This checking is done using essentially the same algorithm as in [CL94].

Consider the dispatch in the expression `aColorPoint1.B_LessOrEqual(aColorPoint2)`. There are three function associations for `B_LessOrEqual`, namely

1. `B_LessOrEqual: T_Real, T_Real → T_Boolean : F_LessOrEqualR`
2. `B_LessOrEqual: T_Point, T_Point → T_Boolean : F_LessOrEqualPoint`
3. `B_LessOrEqual: T_ColorPoint, T_ColorPoint → T_Boolean : F_LessOrEqualColorPoint`

The actual arguments have a product type `T_ColorPoint ⊗ T_ColorPoint`. It is clearly not unifiable with the first association, but it is unifiable with the second and the third. The distance between two product types is calculated as the maximum distance between their respective arguments. The distance between `T_ColorPoint` and `T_Point` is 1 (since `T_ColorPoint` is an immediate subtype of `T_Point`) and the distance between identical types is always 0. Thus, the distance between the actual arguments' type and the arguments' type of the second association is $\max(1, 1) = 1$ and the distance between the actuals and the third association is $\max(0, 0) = 0$. Therefore, the third association is chosen and the function `F_LessOrEqualColorPoint` is called. If we had the expression `aColorPoint.B_LessOrEqual(aPoint)`, it would only be unifiable with the second association, and the function `F_LessOrEqualPoint` would be called.

In this section we have shown that our specification of the problem in Example 3.1 is correct with respect to its intended semantics (disallowing cross-comparisons between different domains, while allowing cross-comparisons between objects that belong to the same domain). It is easy to show that the mechanism described above would correctly typecheck the calls to `B_GreaterOrEqual` and that in all type-correct cases the (anonymous) function associated with this behavior by the parametric type `T_PartiallyOrdered(X)` (Figure 1) would be called. Thus, the specifications of Example 3.1 have been met.

Next we will describe the TIGUKAT behavior type specifications and show that they can be used to correctly type behaviors implementing certain set-oriented operations, such as `B_Map`.

3.3 Behavior types and specifications

Behavior type specifications are important in that they allow correct typing of higher-order functions. This is particularly important in reflexive systems where behaviors and functions are objects. In TIGUKAT, behavior type specifications are written as $A_0, A_1, \dots, A_n \rightarrow R$, where A_0 is the receiver type, A_1, \dots, A_n are argument types, and R is the result type. The above specification, when used as an argument type specification, means that the appropriate argument is a behavior which, when applied to the arguments of

specified types (or their subtypes) yields a result of type R (or a subtype). For example, the behavior B_Map (defined on $\mathbf{T_Set}\langle X \rangle$) has the following specification: $B_Map: x:\mathbf{T_Set}\langle X \rangle, b:(X \rightarrow Y) \rightarrow \mathbf{T_Set}\langle Y \rangle$. This behavior applies the behavior denoted by its argument (b) to every element of the receiver set (x), gathering the results into the set that is returned. It is easy to see that the above type specification is polymorphic and correctly reflects the typing of the behavior B_Map . Polymorphic behavior typing is well-known, but not in the context of a programming language with non-structural user-definable subtyping.

Behavior types in TIGUKAT are behaviors themselves. More precisely, objects representing behavior types are subtypes of both $\mathbf{T_Behavior}$ and $\mathbf{T_Type}$. There is no subtyping between *concrete* behavior types, and types of all³ behaviors in the system are different objects. This is necessary in the context of TIGUKAT which is capable of changing behaviors at run-time to avoid interdependency between behavior definitions. If two behaviors could share a type than a redefinition of one behavior would also affect the other and vice versa.

Behavior types treated as behaviors are defined on $\mathbf{T_Type}$. If B_B is a behavior with n arguments and BT_B is its type, then BT_B is defined as $t_0 : \mathbf{T_Type}, t_1 : \mathbf{T_Type}, \dots, t_n : \mathbf{T_Type} \rightarrow \mathbf{T_Type}$. The semantics of behavior type BT_B is as follows: when applied to types t_0, t_1, \dots, t_n it generates a result type that is a supertype of the type of the result that B_B would have generated if applied to arguments o_0, o_1, \dots, o_n such that type of o_0 is t_0, \dots , type of o_n is t_n . For example, the following expression would generate $\mathbf{T_Boolean}$ when executed: $\mathbf{T_ColorPoint}.BT_LessOrEqual(\mathbf{T_Point})$, and so will the expression $\mathbf{T_ColorPoint}.BT_LessOrEqual(\mathbf{T_ColorPoint})$. From the point of view of the TIGUKAT dispatch mechanism, all regular behavior types are the same and equivalent to $\mathbf{T_RegularBehavior}$, which is their common supertype. Thus, they are in a way similar to light parametric types (Section 3.1).

Theoretically speaking, if we consider behaviors as functions from the domain of objects to the domain of objects, then their types are functions from the domain of types to the domain of types. Both domains are recursive, since behaviors are objects and behavior types are types. Besides, the domain of types is a subdomain of the domain of objects, since types are objects.

This novel treatment of behavior types allows us not only to provide unprecedented run-time type analytic capabilities, but also to use a version of constructive intuitionistic logic to deal with type specifications. We treat type specifications as “types” in Martin-Löf’s type theory [ML84], [NPS90]. A proof of a specification in that theory is a computable expression that we use as a behavior type in TIGUKAT. Unprovable specifications are rejected. Since we have no negation and no existential quantification in the type specification, we use a simple decision procedure which is a variant of [Abr93] with changes proposed in [Dyc92]. In this logic, we also limit weakening to exclude “unsubstantiable” type specifications like $((P \rightarrow Q), X) \rightarrow X$ where we can not find out what P and Q are. Contraction is also limited due to variance specification. For example, the specification $\mathbf{T_Ordered}\langle X \rangle \rightarrow \mathbf{T_Set}\langle X \rangle$ is allowed, while $\mathbf{T_Set}\langle X \rangle \rightarrow \mathbf{T_Ordered}\langle X \rangle$ is not ($\mathbf{T_Ordered}\langle X \rangle$ is nonvariant while $\mathbf{T_Set}\langle X \rangle$ is covariant). Currently we also disallow certain kinds of recursive specifications (for example, $(a : X, b : (X \rightarrow X)) \rightarrow X$). This might change in the future when we know how to compute resulting expressions (in this case, $\Pi(a, b(a), b(b(a)), \dots)$). As far as we are aware, this approach to type specification is novel to our work.

3.4 Union and intersection types

In the context of database programming, it is imperative that a programming language provides union and intersection types. They are required for set-theoretic and relational operations that form the basis for

³Except for the kernel type of regular behavior types $\mathbf{T_RegularBehaviorType}$ that is a type of itself. This is needed to avoid infinite recursion.

declarative SQL-like database querying. It is also important to be able to generate these types at run-time when set-theoretic operations are performed on arguments whose types are not precisely known in advance. TIGUKAT addresses both of these problems by making union and intersection types *atomic*, just like integers and reals. They are there when they are needed, but the system does not keep track of them. This approach also allows us to avoid problems related to their duplication. It is possible because union and intersection types are always *abstract*.

The example of usage of these types is the specification of set-theoretic behaviors in TIGUKAT. For example, B_Union (set union) is defined (on $\mathbf{T_Set}\langle X \rangle$) as $B_Union: \mathbf{T_Set}\langle X \rangle, \mathbf{T_Set}\langle Y \rangle \rightarrow \mathbf{T_Set}\langle X \sqcup Y \rangle$. The operator \sqcup takes the least upper bound of the participating types, while \sqcap takes the greatest lower bound. All bounds always exist due to the atomicity of union and intersection types.

Other set-theoretic behaviors are typed in a similar manner. The combination of union and intersection types and behavior types and specifications in TIGUKAT makes it possible to precisely type the majority of standard relational operations.

3.5 The treatment of assignment

Since TIGUKAT programming language is imperative, assignment is one of its essential features. TIGUKAT assignments are applications of TIGUKAT behaviors; however, these applications are special. Basically, every TIGUKAT behavior can be applied as a *getter* (in the usual way) or as a *setter*⁴ (provided that such application is type correct). When a behavior is applied as a setter, it takes one additional argument (the value being assigned) and produces no result⁵. Note that the additional argument is also used in dispatch. The same is true of TIGUKAT functions. The essential difference between getters and setters appears only at the implementation level since implementation functions for setters and getters are different.

The TIGUKAT system allows the user to specify a function as being **stored** (for example, functions that implement behaviors B_X and B_Y of the type $\mathbf{T_Point}$ in the Figure 1). When such a specification is made, the system automatically creates two implementation functions (a getter and a setter) for the function that is specified as **stored**. It also allocates a slot in the implementation of the appropriate type. In order for this to be possible, the implementation type in question should be an implementation subtype of $\mathbf{IT_Regular}$, since these implementation types are required to provide a variable-length array of slots somewhere in their structure. The most useful feature of stored functions is that they are automatically maintained by the system in case of multiple inheritance when slot numbers change. When this is the case, the system automatically constructs new branches (implementation functions) of the stored function that provide access to the renumbered slots. Stored functions can be named and reused just like any other functions, and all automatically generated branches of the stored function are guaranteed to provide access to the same logical slot. This may be needed if it is necessary to provide a couple of behaviors that access the same logical slot.

Type specification of setters involves the connectives \leftarrow and \leftrightarrow . For example, the behavior B_X of the type $\mathbf{T_Point}$ is defined as $\mathbf{T_Point} \leftrightarrow \mathbf{T_Real}$. This means that B_X produces a result of type $\mathbf{T_Real}$ when applied to the receiver of type $\mathbf{T_Point}$ as a getter and accepts the value of type $\mathbf{T_Real}$ when applied to it as a setter. Function association specifications use \leftarrow and \leftrightarrow in the same manner, thus allowing association of different functions with getter and setter branches of the same behavior.

Theoretically, $X \leftarrow Y$ is treated as $X \rightarrow_s (\mathbf{contravar}Y)$ and $X \leftrightarrow Y$ is treated as $(X \rightarrow_s \mathbf{novar}Y) \wedge (X \rightarrow \mathbf{novar}Y)$ in the type specification logic, since assignment is essentially contravariant on the type of the assigned value.

⁴Also called *acceptor* in some languages, notably BeCecil[CL97].

⁵This is a white lie. The result *is* being produced. It is of type $\mathbf{T_Unit}$. Since $\mathbf{T_Error}$ is a subtype of $\mathbf{T_Unit}$ (as well as of any other type in the system), an error can also be produced and checked for.

The connective \rightarrow_s is analogous to \rightarrow but refers to the setter branch of the behavior type (**s-apply** instead of **apply** in the resulting expression). Thus, each behavior type can be thought of as a pair of two (getter type and setter type) functions.

There is an additional constraint related to the relationship between getter and setter type branches of the same behavior B_B . It states that for any type X , $X._s BT_B \preceq X.BT_B$, where $._s$ denotes the application of the setter-type branch of the behavior type. The meaning of this constraint is that one is not allowed to set something that one will be unable to get. Since this constraint is non-monotonic, it requires checking of all subtypes of a given type. The system checks this automatically, as a part of behavior consistency check.

The syntax for setter applications is $r.B_B(a) := v$, which calls the setter branch of the behavior B_B with argument tuple (r,a,v) . Thus, an assignment to the X coordinate of a point would be written as $aPoint.B_X := 1.0$, a familiar notation for assignment.

A special (heavy) parametric type $\mathbf{T_Var}\langle\mathbf{novar}\ X\rangle$ provided by the system makes it easier to deal with mutable container types. The type $\mathbf{T_Var}\langle X\rangle$ makes sense for all types X except for types of the form $\mathbf{T_Var}\langle X\rangle$, $\mathbf{T_Null}$ and $\mathbf{T_Error}$ ⁶. For any type X , $\mathbf{T_Var}\langle X\rangle \preceq X$. Thus, all operations that are defined for type X are also defined for type $\mathbf{T_Var}\langle X\rangle$. $\mathbf{T_Var}\langle X\rangle$ also defines the behavior $B_Get: \mathbf{T_Var}\langle X\rangle \leftrightarrow X$. The expression $x.B_Get := v$ can be abbreviated to $x := v$. There also exists the behavior $B_Destructive: (X, Y \rightarrow X) \rightarrow (\mathbf{T_Var}\langle X\rangle, Y \rightarrow)$ that can be used to define "destructive" versions of existing behaviors for \mathbf{Var} -types⁷. For example, objects of types $\mathbf{T_Set}\langle X\rangle$ are immutable and the behavior $B_Intersect: \mathbf{T_Set}\langle X\rangle, \mathbf{T_Set}\langle Y\rangle \rightarrow \mathbf{T_Set}\langle X \sqcup Y\rangle$ produces a new set each time it is applied. If we have an object $mSetX$ of type $\mathbf{T_Var}\langle \mathbf{T_Set}\langle X\rangle\rangle$ (type of mutable sets) and an object $SetY$ of type $\mathbf{T_Set}\langle Y\rangle$, we can write $mSetX.(B_Union.B_Destructive)(SetY)$ and the result of the execution of this expression would be the placement of the intersection of the two sets inside $mSetX$, replacing its previous contents.

This concludes our overview of the TIGUKAT type system. We have seen that it provides powerful parametric types, parametric behaviors, precise typing of behavior and assignments, and reflexive capabilities. Next, we will briefly discuss the implementation type system and the class system of TIGUKAT.

4 Implementation type system

TIGUKAT separates interface from the implementation. Since types in TIGUKAT correspond to interface specifications, a concept of implementation specification is needed. This concept is provided in the form of implementation types.

Implementation types are structure definitions in TIGUKAT. They can be compared to the structure definitions in C. Implementation types form a tree hierarchy (single inheritance) under the root implementation type $\mathbf{IT_OID}$. This hierarchy is totally independent of the TIGUKAT type hierarchy. Type $\mathbf{IT_Regular}$ is the type of all regular TIGUKAT objects. Its implementation subtypes must provide storage for variable-length array of slots. An implementation type also defines low-level special implementation functions for object storage/retrieval from the database, creation, deletion, and several others. Ordinary users should not be allowed to define new implementation types, modify old ones, or supply their own implementation functions, since it is a low-level, potentially dangerous activity. On the other hand, a database administrator can use implementation types not only to optimize particular types, but also to provide interoperability by defining implementation types of external (non-TIGUKAT) objects and the implementation functions dealing with them. Once this is done, a uniform TIGUKAT interface to such objects can be easily provided.

⁶For a type E for which it does not make sense, $\mathbf{T_Var}\langle E\rangle \equiv \mathbf{T_Error}$.

⁷The code for this behavior is `return $\lambda x,y.\{x := x.\mathbf{self}(y)\}$.`

Implementation functions are pieces of low-level (currently C++ with certain restrictions) code that are optimized to deal with a particular implementation. The implementation function is chosen by the second phase of the TIGUKAT dispatch according to the implementation types of the arguments. An example specification of implementation functions and types is given in Figure 1, where integer and real numbers take advantage of the optimized implementation of the function *F_LessOrEqualR*.

5 Class system

Classes in TIGUKAT are used to maintain extents of types and to automatically group objects of the same type. Object creation in TIGUKAT occurs only via classes (for example, to create an object of type **T_Person**, one would write **C_Person.B_New**), and every TIGUKAT object is an instance of a class. A type can have no more than one class; at the same time, several types can share a class. Classes can be *abstract* (of types **T_AbstractClass** $\langle X \rangle$ like **C_Object**), *infinite* (of types **T_InfiniteClass** $\langle X \rangle$ like **C_Integer**), *finite* (of types **T_FiniteClass** $\langle X \rangle$ like **C_Boolean**), and *regular* (of types **T_Class** $\langle X \rangle$ like **C_Person**). All class types are heavy parametric types and are therefore represented by regular objects in the system. Infinite, finite, and regular classes are called *concrete classes* since they can have instances. Regular classes allow object creation. If a class has many implementation types, a particular one has to be chosen when an object is created. Classes are objects; they map to subtypes of a designated type **T_Class**. Classes have a partial order (subclassing) defined on them. Subclassing on classes corresponds to set inclusion on their deep extents and is parallel to subtyping. *The shallow extent* of a class is defined as the set of all objects that map to a type associated with the class, while *the deep extent* of a class is defined to be the set of all objects that map to types associated with this class and all of its subclasses. Thus, every object in TIGUKAT is an instance of its class (the class related to the type the object maps to) and belongs to the shallow extent of this class only. However, it also belongs to deep extents of superclasses of this class. Classes are extensively used in database querying (for example, a query "Find all persons younger than 35" can be expressed as **C_Person.B_DeepExtent.B_Select** $(\lambda x. \{x.B_Age.B_LessOrEqual(35)\})$, where **C_Person** is the class of persons).

TIGUKAT classes also play a major role in dynamic schema evolution. All checks for behavior and function completeness and consistency are done when a type is associated to a concrete class or when a change in behavior or function specification is propagated through a type associated with a concrete class.

An example of class specification (in this case embedded in a type specification) is given in Figure 1. Since classes are objects, behaviors defined on their types can be applied to them. These behaviors can be defined in the definition of the class itself (as behavior *B_Max* is defined in the definition of the class **C_Real**). These behaviors correspond to class methods in Smalltalk [GR89] or static member functions in C++ [Str91]. Since class types are nonvariant, the class behaviors in TIGUKAT are *not* inherited. This is done because class behaviors are mostly used for object creation, and the object creation tends to be nonvariant since the creation of objects of subtypes usually requires more arguments than the creation of objects of the supertypes. Future research may provide some combination of covariance and contravariance for classes.

Classes in TIGUKAT constitute a bridge between types and implementation types, just like functions provide a bridge between behaviors and implementation functions. Thus access to classes by ordinary users must be limited. Internally, however, classes play a major role as directors of schema change propagation and internal checking. They are also frequently used in database querying because of their ability to automatically maintain the extents of types.

6 Reflexivity, uniformity, and schema evolution

The possibility of dynamic schema evolution is one of the main ideas behind the development of TIGUKAT. Everything that can be done in TIGUKAT statically can also be done dynamically. For instance, all types, behaviors, classes, etc related to the specification of Example 3.1 could be created dynamically, using a series of behavior applications. The specifications in Figure 1 are no more than a series of object specifications. Those objects, like any TIGUKAT objects, could have been created and modified dynamically to yield the same schema.

The reflexivity and uniformity of the system allow us not only to dynamically query and/or modify a database schema, but also to write generic database programs such as browsers. Since every TIGUKAT object knows its type at run-time and since that type is a TIGUKAT object as well, a program can “learn” type information at run-time and change its behavior according to it. This is possible in the presence of static type-checking because of the presence of behaviors such as *BA*s (Section 3.1) that allow statically checkable dynamic type exploration.

7 Related work

The recent advances in the development of object-oriented type systems have produced significant research output. It is therefore virtually impossible to list all significant results in the area without writing a full-sized survey. Therefore here we will concentrate on the research that is specifically aimed at programming languages rather than at their theoretical basis or implementation aspects.

The first list of requirements to the type systems of the database programming languages was compiled in [AB87]. Then came the Manifesto [ABD⁺92] specifically aimed at object-oriented databases. The latest is [AM95] describing requirements for orthogonally persistent object systems. Open problems in object-oriented type systems were also listed in [BP94] which was an inspiration for the authors. Another discussion of problems in object-oriented programming languages appears in [L95].

Using product types for method dispatch was reported in [Age95] and has been shown to be useful. A variant of it has also been proposed in [FE96]. Typing and type checking of multi-methods was also considered in [ADL91], [Ghe91], [MHH91], [BG95], and [CL94]. Certain related problems were also addressed in [BC95]. [Cas95] and [CGL95] develop the theoretical account of multi-methods and product types.

Problems related to the specification of binary methods (analogous to *B_LessOrEqual* from the Example 3.1) were discussed in [BCC⁺96]. The solution presented here is different from the ones proposed in that paper.

One of the first object-oriented database programming languages was O++ [AG89]. It is a persistent dialect of C++ [Str91] and thus inherits all deficiencies of its type system: the absence of separation between interface and implementation (both specification and inheritance are coupled), limited support for method typing, the absence of reflexivity and schema evolution, single dispatch, and very limited parametricity. One of its strengths, of course, is its close relationship with C++; however, in terms of its type system O++ is rather limited.

Another database programming language based on C++ is E [RCS93]. It improves the handling of parametric types, but otherwise inherits the same C++ drawbacks as O++.

Completely different approach was taken in CLOS [BDG⁺88]. CLOS is reflexive and has multiple dispatch; however, it lacks static type checking.

Dylan [App94] is more recent language based (to a degree) on CLOS, but with static typing. It has multi-methods and separates interface from implementation, is reflexive but lacks precise type checking of

behaviors and parametricity.

Another early effort in the area is presented by the database programming language Galileo [ACO85]. It is based on ML [MTH90] and has a functional flavor. Galileo is reflexive, but its type system is also limited: parametric types are absent, and the dispatch is non-existent since the language is not object-oriented.

Amber [Car86] is also ML-based. It has structural subtyping and is partially reflexive since it is able to examine (but not change) types at run-time. Amber is not object-oriented and does not provide clear distinction between interface and implementation.

Machiavelli [OBBT89], [PA96] is also a database programming language based on ML. It has impressive type system specifically designed to support database applications complete with union, intersection, and functional types. However, it is not object-oriented, so the issues of inheritance and dispatch do not arise there. Reflexivity of Machiavelli is also limited.

Another recent successor of Galileo is the database programming language Fibonacci [AGO95]. It supports static typing, is reflexive, and partially separates interface from implementation. It also has parametric system-defined types, but the user can not create new ones. Fibonacci focuses on *roles* that represent different aspects of the same object. It also has single dispatch and does not support precise function typing.

Napier88 v.2.2.1 [MBC⁺96] is an imperative database programming language with very powerful type system and certain reflexive capabilities. It is not object-oriented and its parametric types though powerful can not be used until fully instantiated.

The language TM [BBZ93], [BBB⁺93] is partially reflexive and separates interface from implementation. However, it does not allow method redefinitions, does not have multiple dispatch, does not support user-definable parametric types, and its subtyping does not imply substitutability.

TL [MMS94] is a database programming language based on ML. It has a very powerful type system, but lacks object-oriented features. In addition to that TL does not directly support interface inheritance.

The language PolyTOIL [BSG95], [BSG94] has one of the most impressive type systems. It has multiple dispatch, user-definable parametric types, function type specifications, separates interface from implementation, and has a static type checking. However, since it is not a database programming language, it lacks reflexivity and support for union and intersection types needed in the context of database programming.

Cecil [Cha93] is delegation-based. It has suggestive type checking, but its type system is quite impressive. It includes parametric types, multiple dispatch, separation of interface and implementation. It is also partially reflexive (the types are not part of the game, they are merely annotations). Dispatch in Cecil is done on representations and does not involve types. Besides, Cecil was not designed to deal with database applications and query typing in Cecil is problematic.

Another imperative object-oriented programming language is Sather [SOM93]. Sather has parametric types and function types, but does not have multiple dispatch. Its separation between interface and implementation, although present, is limited.

Theta [MRBC95] is an object-oriented programming language with its type system specifically designed to support parametric polymorphism. Although impressive, it lacks the ability to deal with non-covariant parametric type specifications.

BeCecil [CL97], being a successor of Cecil, has a very powerful type system. It supports multiple dispatch, separates interface from implementation, and is partially reflexive. It also supports acceptors similar to setter behavior branches in TIGUKAT (Section 3.5). However, BeCecil does not have parametric types, union and intersection types. Its types are not objects in the language and are not used for dispatch.

Having reviewed the existing languages we can conclude that none of them has the full set of features of the TIGUKAT language, namely powerful parametric polymorphism, inclusion polymorphism (subtyping), precise behavior typing, complete separation between interface and implementation, union and intersection

types and reflexivity. All these features are considered essential for a type system of a modern object-oriented database programming language according to the papers [ABD⁺92], [AM95], and [BP94].

8 Conclusions

In this paper we have presented the type system of the TIGUKAT database programming language. We have also illustrated it by a complicated modeling example and have shown how such an example can be programmed in TIGUKAT. By comparing TIGUKAT to the existing programming languages we have demonstrated that the set of essential features the TIGUKAT type system possesses is not matched or superseded by any of these languages.

Future research directions include complete system implementation, further development of type specification logic, development of the module and name space mechanism, complete specification of the programming language, and research dealing with the persistence model and schema evolution in TIGUKAT.

References

- [AB87] Malcolm P. Atkinson and O. Peter Buneman, *Types and persistence in database programming languages*, ACM Computing Surveys **19** (1987), no. 2, 105–190.
- [ABD⁺92] Atkinson, Banchilon, DeWitt, Dittrich, Maier, and Zdonik, *The object-oriented database system manifesto*, Building an Object-Oriented Database System: The Story of O_2 (François Banchilon, Claude Delobel, and Paris Kanellakis, eds.), 1992.
- [Abr93] Samson Abramsky, *Computational interpretations of linear logic*, Theoretical Computer Science **111** (1993), no. 1–2, 3–57.
- [ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini, *Galileo: A strongly-typed, interactive conceptual language*, ACM Transactions on Database Systems **10** (1985), no. 2, 230–260.
- [ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay, *Static type checking of multi-methods*, ACM SIGPLAN Notices **26** (1991), no. 11, 113–128, In Proc. of OOPSLA'91.
- [AG89] R. Agrawal and N. H. Gehani, *ODE (object database and environment): The language and the data model*, Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data, 1989, pp. 36–45.
- [Age95] Ole Agesen, *The cartesian product algorithm*, Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95) (Åarhus, Denmark) (W. Olthoff, ed.), LNCS 952. Springer, August 1995.
- [AGO95] Antonio Albano, Giorgio Ghelli, and Renzo Orsini, *Fibonacci: A programming language for object databases*, VLDB Journal **4** (1995), 403–444.
- [AM95] Malcolm Atkinson and Ronald Morrison, *Orthogonally persistent object systems*, VLDB Journal **4** (1995), no. 3, 319–401.
- [App94] Apple Computer, Inc., *Dylan interim reference manual*, 1994.
- [BBB⁺93] René Bal, Herman Balsters, Rolf A. De By, Alexander Bosschaart, Jan Folkstra, Maurice Van Keulen, Jacek Skowronek, and Bart Termorshuizen, *The TM manual*, December 1993, Version 2.0 revision c.

- [BBZ93] Herman Balsters, Rolf A. De By, and Roberto Zicari, *Typed sets as a basis for object-oriented database schemas*, Proceedings ECOOP'93 (Kaiserslautern, Germany) (O. Nierstrasz, ed.), Springer-Verlag, July 1993, pp. 161–184.
- [BC95] John Boyland and Giuseppe Castagna, *Type-safe compilation of covariant specialization: A practical case*, Tech. Report UCB/CSD-95-890, University of California, Computer Science Division (EECS), Berkeley, California 94720, November 1995.
- [BCC⁺96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce, *On binary methods*, Theory and Practice of Object Systems (1996).
- [BDG⁺88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon, *Common Lisp Object System specification*, June 1988, X3J13 Document 88-002R.
- [BG95] Elisa Bertino and Giovanna Guerrini, *Objects with multiple most specific classes*, ECOOP'95, 1995.
- [BP94] Andrew Black and Jens Palsberg, *Foundations of object-oriented languages*, ACM SIGPLAN Notices **29** (1994), no. 3, 3–11, Workshop Report.
- [BSG94] Kim B. Bruce, Angela Schuett, and Robert Van Gent, *A type-safe polymorphic object-oriented language*, Accessible by anonymous FTP, July 1994.
URL: <ftp://cs.williams.edu/pub/kim/PolyTOIL.dvi>
- [BSG95] Kim B. Bruce, Angela Schuett, and Robert Van Gent, *PolyTOIL: A type-safe polymorphic object-oriented language*, Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95) (Åarhus, Denmark) (W. Olthoff, ed.), LNCS 952. Springer, August 1995, Extended abstract.
- [Car86] Luca Cardelli, *Amber*, Combinators and Functional Programming Languages (G. Cousineau, P-L. Curien, and B. Robinet, eds.), Springer-Verlag, 1986, LNCS 242.
- [Cas95] Giuseppe Castagna, *A meta-language for typed object-oriented languages*, Theoretical Computer Science **151** (1995), no. 2, 297–352.
- [CGL95] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo, *A calculus for overloaded functions with subtyping*, Information and Computation **117** (1995), no. 1, 115–135.
- [Cha93] Craig Chambers, *The Cecil language: Specification and rationale*, Tech. Report TR 93-03-05, Department of Computer Science and Engineering, FR-35, University of Washington, March 1993.
- [CL94] Craig Chambers and Gary T. Leavens, *Typechecking and modules for multi-methods*, ACM SIGPLAN Notices **29** (1994), no. 10, 1–15, In Proc. of OOPSLA'94.
- [CL97] Craig Chambers and Gary T. Leavens, *BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing*, FOOL 4, The Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, France, January 1997.
- [Dyc92] Roy Dyckhoff, *Contraction-free sequent calculi for intuitionistic logic*, The Journal of Symbolic Logic **57** (1992), no. 3, 795–807.
- [FE96] Barrett M F and Giguere M E, *A note on covariance and contravariance unification*, ACM SIGPLAN Notices **31** (1996), no. 1, 32–35.

- [Ghe91] Giorgio Ghelli, *A static type system for message passing*, ACM SIGPLAN Notices **26** (1991), no. 11, 129–145, In Proc. of OOPSLA'91.
- [GLÖS95] Iqbal A. Goralwalla, Yuri Leontiev, M. Tamer Özsu, and Duane Szafron, *A uniform behavioral temporal object model*, Tech. Report TR 95-13, Department of Computing Science, University of Alberta, July 1995.
- [GR89] A. Goldberg and D. Robson, *ST-80, the language*, Addison-Wesley, 1989.
- [L95] Madsen O L, *Open issues in object oriented programming: A scandinavian perspective*, Software Practice and Experience **25 Supplement** (1995), 3–43.
- [LML⁺94] V. G. Luty, A. B. Merkov, Y. V. Leontiev, E. J. Gawrilow, N. A. Ivanova, M. E. Iofinova, M. L. Paklin, and A. K. Hodataev, *DBMS Modula-90K*, RAN Data Processing Center, Moscow, 1994, /in Russian: Sistema Programmirovaniya Baz Dannyh Modula-90K/.
- [MBC⁺96] Ron Morrison, Fred Brown, Richard Connor, Quintin Cutts, Alan Dearle, Graham Kirbi, and Dave Munro, *Napier88 reference manual*, University of St. Andrews, March 1996, Release 2.2.1.
- [MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking, *Multi-methods in a statically-typed programming language*, ECOOP'91 (Geneva, Switzerland) (Pierre America, ed.), July 1991, LNCS 512, pp. 307–324.
- [ML84] Per Martin-Löf, *Intuitionistic type theory*, Bibliopolis, Napoli, 1984.
- [MMS94] Florian Matthes, Sven Müßig, and J. W. Schmidt, *Persistent polymorphic programming in Tycoon: An introduction*, FIDE Technical Report Series FIDE/94/106, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.
- [MRBC95] Day M, Gruber R, Liskov B, and Myers A C, *Subtypes vs where clauses: Constraining parametric polymorphism*, SIGPLAN Notices **30** (1995), no. 10, 156–168.
- [MTH90] Robin Miller, Mads Tofte, and Robert Harper, *The definition of Standard ML*, MIT Press, 1990.
- [NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith, *Programming in Martin-Löf's type theory: An introduction*, Clarendon Press, Oxford, 1990.
- [OBBT89] Atsushi Ogori, Peter Buneman, and Val Breazu-Tannen, *Database programming in Machiavelli — a polymorphic language with static type inference*, SIGMOD Record **18** (1989), no. 2, 46–57.
- [ÖMS95] M. T. Özsu, A. Muñoz, and D. Szafron, *An extensible query optimizer for an objectbase management system*, Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM'95), Baltimore, October 1995, pp. 188–196.
- [ÖPS⁺95] M. T. Özsu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Muñoz, *TIGUKAT: A uniform behavioral objectbase management system*, The VLDB Journal **4** (1995), no. 3.
- [PA96] Buneman P and Ogori A, *Polymorphism and type inference in database programming*, ACM Transactions on Database Systems **21** (1996), no. 1, 30–76.
- [PÖ95a] Randal J. Peters and M. Tamer Özsu, *An axiomatic model of dynamic schema evolution in objectbase management systems*, Tech. Report TR 95-02, Department of Computer Science, University of Manitoba, April 1995.
- [PÖ95b] Randal J. Peters and M. Tamer Özsu, *Axiomatization of dynamic schema evolution in objectbases*, Proceedings of the 11th International Conference on Data Engineering, March 1995.
- [PSVS95] Buneman P, Naqvi S, Tannen V, and Wong L S, *Principles of programming with complex objects and collection types*, Theoretical Computer Science **149** (1995), no. 1, 3–48.

- [RCS93] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh, *The design of the E programming language*, ACM Transactions on Programming Languages and Systems **15** (1993), no. 3, 494–534.
- [SOM93] Clemens Szypersky, Stephen Omohundro, and Stephan Murer, *Engineerig a programming language: The type and class system of Sather*, Tech. Report TR-93-064, The International Computer Science Institute, November 1993.
- [Str91] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1991.