Orthogonal to the Java Imperative*

Suad Alagić, Jose Solorzano, and David Gitchell

Department of Computer Science, Wichita State University
Wichita, KS 67260-0083, USA
alagic@cs.twsu.edu

Abstract. Three nontrivial limitations of the existing Java^{TM} technology are considered from the viewpoint of object-oriented database technology. The limitations are: lack of support for orthogonal persistence, lack of parametric (and in fact bounded and F-bounded) polymorphism and lack of an assertion (constraint) language. These limitations are overcome by leaving Java as it is, and developing a declarative (query in particular) component of the Java technology. This declarative language is implemented on top of the Java Virtual Machine, extended with orthogonal and transitive persistence. The model of persistence also features complex name space management.

Keywords: Declarative languages, orthogonal persistence, F-bounded polymorphism, Java Virtual Machine.

1 Introduction

In spite of the fact that Java^{TM} [17] is a product of good language design, it exhibits several serious limitations with respect to object-oriented database management. A perfectly valid justification is that it was probably never conceived as a database programming language. Yet, some of these database-critical features are important for object-oriented programming in general.

The first such limitation is, of course, the lack of support for persistence. Persistence is a key feature of object-oriented database technology [10].

The second limitation is the absence of parametric polymorphism. This limitation is particularly critical in database management because most database technologies rely heavily on collections. Static typing of collections in Java is impossible, and so is static typing of queries. Awkward type casts and extensive dynamic type checks are required, affecting both efficiency and reliability.

The third limitation is the lack of an assertion language. Such declarative capabilities for expressing preconditions, postconditions and class invariants are important for object-oriented programming in general ([24], [22], [25]). Moreover, constraints and other declarative language features (including queries) are critical for any viable database technology.

* This material is based upon work supported in part by the U.S. Army Research Office under grant no. DAAH04-96-1-0192.

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 212–233, 1998.

Several projects are under way, extending the Java technology with persistence [11], parametric polymorphism ([27], [9]) and even queries [15].

The projects extending the language with parametric polymorphism necessarily require changing the language, but they do not necessarily change the Java Virtual Machine.

Java OQL [15] does not change the language, but supporting queries as methods and as objects requires run-time reflection [3] where the standard notion of static typing is extended to dynamic compilation. In addition, persistence capabilities are required in the underlying architecture.

PJama (formerly PJAVA) [11] provides orthogonal persistence without changing the language, but it requires an extension of the Java Virtual Machine. The Java Virtual Machine is specified in [23].

The approach that we take in extending the Java technology with database capabilities is different from any of the above. We do not change the language in spite of its limitations. We leave Java as it is. But we develop a declarative language with all the desired features discussed above and make it fit into the existing Java technology. An extension of the Java Virtual Machine is still required in order to support a sophisticated and orthogonal model of persistence in order to implement this declarative language.

The declarative language *MyT* presented in this paper makes it possible to express preconditions, postconditions and class invariants. This addresses the problem of the lack of assertions in Java. But the constraint language goes further in allowing declarative specification of methods.

The constraint language has its limitations in expressiveness ([2], [4]). Just like data languages, it is not computationally complete. But it covers a variety of non-trivial applications [6] and it is integrated into the Java programming environment by allowing references to Java classes.

On the other hand, Java classes can access persistent objects and their classes, which are created by *MyT*. Java classes can also naturally create persistent classes and objects without any references to *MyT* facilities. This is done by simply relying on PJama, and possibly on our name space management extension.

Java is not a persistent programming language. *MyT* features an orthogonal model of persistence and transitive persistence (persistence by reachability). Unlike all other approaches, persistence capabilities in this model are associated with the root (top) class. This way all classes are persistence capable. The model is naturally based on reachability. In addition, this model of persistence features hierarchical name space management, extending the existing Java mechanism based on packages. Such complex name space management is lacking in persistent supporting Java extensions, such as PJama [11].

Collection classes are critical for object-oriented database technologies. Typing generic collection classes in Java has well-known problems. The best we can do is to specify generic collection classes whose elements are of type *Object*. Accessing such collections requires type casts, which means dynamic type checks.

Thus static type checking of typical declarative constructs, queries in particular, cannot be performed in the Java type system.

Because of this, the constraint language supports parametric polymorphism, thus overcoming this serious drawback of Java in database applications [3]. The form of parametric polymorphism is F-bounded [14], so it allows proper typing of ordered collections and indices ([8], [3]).

Unlike the existing Java preprocessor systems, this system is a real compiler that generates Java byte code. In addition, a real novelty of this compiler is that it generates methods from declarative constraints. This in particular includes proper treatment of preconditions, postconditions and class invariants in the generated code.

A class in the declarative constraint language is translated into a Java class file. This must be done in such a way that the generated file passes the Java verification test. This test was designed to recognize class files that are generated by a possibly hostile compiler. This is precisely one of the reasons why several Java extending projects are based on preprocessors. Our compiler generates Java class files directly.

The constraint language naturally supports queries. Queries refer to persistent collections. They can be statically type checked, which would not be possible in the type system underlying Java or Java OQL ([15], [3]). The run-time support includes algorithms that operate on the complex implementation architecture for persistent collections. This architecture includes access paths implemented via extendible hashing, or as indices.

The paper is organized as follows. In section 2 we introduce the constraint language, its logic basis, collection classes and queries. In section 3 we introduce advanced typing techniques available in MyT (bounded and F-bounded type quantification). In section 4 we introduce our model of persistence, and name space management as supported by environments. In section 5 we describe the underlying persistence implementation architecture. In section 6 we describe our implementation techniques for F-bounded polymorphism, along with the specifics of Java class file representation. Section 7 deals with compilation issues, which include proper management of Java class files and Java byte code generation from the constraint language.

2 The Constraint Language

2.1 The Logic Basis

The declarative language MyT has a logic basis that is in fact temporal [4]. This point is not particularly significant for the main results of this paper. Other logic basis are also possible, and the results would apply to many of those. However, the logic basis should be able to express typical types of assertions in object-oriented programming, such as preconditions, postconditions, class invariants, history properties, etc.

The underlying temporal paradigm is intentionally simple. It is based on the notion of time, that is discrete and linear, extending infinitely to the future.

This paradigm and the full-fledged temporal constraint language are elaborated in full detail in separate papers ([2], [4]).

Illustrative examples of classes presented in this paper make use of only two temporal operators. The operator *always* is denoted as $\Box$. If $\mathcal{C}$ is a constraint, then $\Box\mathcal{C}$ is true in the current state iff $\mathcal{C}$ evaluates to true in all object states, starting with the current one. The operator *nexttime* is denoted as $\bigcirc$. The constraint $\bigcirc\mathcal{C}$ is true in the current state iff the constraint $\mathcal{C}$ is true in the next object state.

The constraints are expressed by temporal Horn clauses. Standard Horn clauses have the form $A \leftarrow B_1, B_2, ..., B_n$, where A is the head, $B_1, B_2, ..., B_n$ is the body of the clause, $A, B_1, B_2, ..., B_n$ are atomic predicates, $\leftarrow$ denotes implication and comma denotes conjunction. The constraint language, in addition, allows the temporal operators to appear in temporal Horn-clauses.

There are restrictions on the usage of the temporal operators in the constraint language. These rules limit the expressive power of the language, but at the same time they guarantee the existence of the execution model of the language, and its formal semantics [5]. In spite of its limitations, the language allows a wide variety of database oriented applications to be handled in a high-level, declarative manner.

2.2 Collection Classes

The constraint language will be illustrated by samples of collection classes. The class *Collection* given below is equipped with a predicate method *belongs* indicating whether its argument belongs to a collection object, the receiver of the message. Messages of this type are called *observers*. The corresponding Java term is *accessor methods*. These messages allow inspection of the underlying hidden object state.

The *Collection* class is equipped with two mutator methods *insert* and *delete* that change the underlying object state. The constraint section specifies the effect of the mutator messages on the only observer of the collection object state. The constraints illustrate how *preconditions* and *postconditions* are specified in the constraint language.

Class Collection[T]
Observers
belongs(T)
Mutators
insert(T); delete(T)
Constraints
ForAll X:T;
$\Box$(self.belongs(X) $\leftarrow$ self.delete(X)),
$\Box$($\bigcirc$self.belongs(X) $\leftarrow$ self.insert(X))
End Collection.

The *precondition* for deletion of an element X is that the element X belongs to the collection, hence the constraint $\Box$(self.belongs(X) $\leftarrow$ self.delete(X)). The

postcondition for inserting X into the collection is that after the insertion X belongs to the collection. This is expressed using the next state operator ◯ in the constraint □(◯self.belongs(X) ← self.insert(X)). The always operator □ is used because these constraints apply to all object states.

Consider now a class *Bag* derived by inheritance from the class *Collection*. *Bag* has an additional observer method *occurs* which specifies how many times the first argument of this method belongs to the underlying *Bag* object. In addition, *Bag* introduces two constructor methods *union* and *intersect*.

The *initial constraint* new(Bag[T]).occurs(X,0) ← specifies the initial state of a bag object when its is created by a new message.

The constraint $\Box(self.belongs(X) \leftarrow self.occurs(X, N), N.gtr(0))$ is a *class invariant*. It involves the observer methods only. It applies to all object states and thus the only temporal operator used is always.

The *transition constraints* □(◯self.occurs(X, N.succ()) ← self.occurs(X, N), self.insert(X)) and □(◯self.occurs(X, N.pred()) ← self.occurs(X, N), self.delete(X)) specify the effect of the inherited mutators *insert* and *delete* on the newly introduced observer *occurs*.

Finally, the *constructor constraints* □(self.union(B).occurs(X, M.max(N)) ← self.occurs(X, M), B.occurs(X,N)) and □(self.intersect(B).occurs(X, M.min(N)) ← self.occurs(X, M), B.occurs(X, N)) specify the observer *occurs* of the bag object constructed by the *union* and *intersect* methods.

Class Bag[T]
Inherits Collection
Observers
occurs(T, Natural)
Constructors
union(Bag[T]): Bag[T],
intersect(Bag[T]): Bag[T]
Constraints
ForAll B: Bag[T]; X:T; M, N: Natural;
new(Bag[T]).occurs(X,0) ←,
□(self.belongs(X) ← self.occurs(X, N), N.gtr(0)),
□(◯self.occurs(X, N.succ()) ← self.occurs(X, N), self.insert(X)),
□(◯self.occurs(X, N.pred()) ← self.occurs(X, N), self.delete(X)),
□(self.union(B).occurs(X, M.max(N)) ← self.occurs(X, M), B.occurs(X,N)),
□(self.intersect(B).occurs(X, M.min(N)) ← self.occurs(X, M), B.occurs(X, N))
End Bag.

2.3 Queries

It should come as no surprise that queries are objects in this system. By way of comparison, incorporating OQL query capabilities into Java comes with nontrivial problems [3]. These problems are manifested by Java OQL [15], which requires run-time reflection (or dynamic compilation) as an implementation technique.

The type of reflective techniques required for a type safe implementation of Java OQL have been developed for other persistent object systems ([16], [12]).

But these techniques are by no means trivial, and go beyond the bounds of the usual static type checking techniques. *MyT* queries can be statically type checked, as one would naturally expect.

The core of the query capabilities is the predefined parametric class Query[T] equipped with an observer *qualification* and a constructor *eval*. The latter takes the queried collection as its argument and produces the collection which represents the result of the query. The objects that appear in the result of the query are those that satisfy *qualification*. This is easily expressed in the constraint section of this class.

Class Query[T]
Observers
qualification(T)
Constructors
eval(Collection[T]): Collection[T]
Constraints
ForAll X: T; S: Collection[T];
□(self.eval(S).belongs(X) ← S.belongs(X), self.qualification(X))
End Query.

A specific query is obtained by instantiating the above class, as in the example given below:

Class MyQuery[Employee]
Inherits Query[Employee]
Constraints
ForAll X: Employee; N: Real;
□(self.qualification(X) ← X.job("programmer"), X.salary(N), N.gtr(70,000.00))
End MyQuery.

3 The Type System

The type system consists of three levels. The third (topmost) level consists of a single object *Class*. Instances of *Class* are classes. These are second-level objects. Instances of classes are first-level objects. This view fits the Java type system. In addition, it fits the orthogonal model of persistence which supports reachability.

The formal development of the type system is not the topic of this paper. It is given in a separate paper [4] together with a proof of its type safety. The type system of full-fledged *MyT* is based on self types ([1], [13]). The type system underlying this paper is more restrictive in order to make it compatible with the Java type system. Self types and multiple dispatch required for their type-safe implementation are thus omitted.

3.1 Bounded Type Quantification

A simple form of parametric polymorphism used in the above sample collection classes does not cover very important situations required in most database

technologies. Queries often produce ordered collections and query evaluation algorithms often require ordered collections. A typical access path used in query evaluation is an index. Persistent collections in database systems thus may be ordered and equipped with indices. Proper typing of such collections requires bounded type quantification (constrained genericity in Eiffel ([24], [25])) and even F-bounded type quantification ([14], [8], [3]) in the type parameter section of a class.

An ordered collection may be specified using bounded type quantification as follows:

Class OrderedCollection[T **Inherits** Ordered]
Inherits Collection[T]
...
End OrderedCollection.

The class *Ordered* used to constraint the type parameter of the class *OrderedCollection* is given below:

Class Ordered
Observers
lessThan(Any)
...
End Ordered.

3.2 F-Bounded Polymorphism

Using *Any* as the type of the argument of the ordering method *lessThan* obviously leads to dynamic type checking. Static typing of ordered collections is accomplished by making the class *Ordered* parametric:

Class Ordered[T]
Observers
lessThan(T)
...
End Ordered.

An ordered collection may now be defined using F-bounded polymorphism as follows:

Class OrderedCollecton[T **Inherits** Ordered[T]]
Inherits Collection[T]
...
End OrderedCollection.

Proper and static typing of the *Index* class involves both bounded and F-bounded type quantification [3].

Class Index[T0 **Inherits** Ordered[T0], T **Inherits** T0]
Constructors
select(T0): Collection[T]
...
End Index.

In the above class $T0$ denotes the type of the search attributes. $T0$ must be equipped with an ordering method *lessThan*, hence the F-bounded condition $T0$ **Inherits** $Ordered[T0]$. T stands for the type of elements of the indexed collection, hence the bounded condition T **Inherits** $T0$. The constructor *select* returns the set of all objects of the underlying collection with the given value of the search attributes supplied as the argument of the *select* method.

A technique related to F-bounded polymorphism is called matching [1], and it is perhaps more intuitive and easier to grasp. It should be noted that these more advanced typing notions come with a price. The underlying type system becomes more complex [18], and it may even be undecidable [28].

4 Persistence

4.1 Model of Orthogonal Persistence

The underlying model of persistence has the following fundamental properties:

- *Persistence is orthogonal to the class hierarchy.* Objects of any level and of any type may be persistent.
- *Persistence is transparent to clients.* This means that clients do not move objects to and from stable storage, nor do they deal with files, etc.
- *Persistence is per-object based.* Objects of a particular class may be persistent or transient, and in fact there may be in general objects of both kinds for a given class.
- *Persistence is based on message-passing.* An object is promoted to persistence by sending a *persists* message to that object.
- *Persistence is complete with respect to observable properties of objects.* This property is known as *reachability* or *transitive persistence*. It guarantees that all observable properties of a persistent object are well-defined even though they may in fact refer to other objects.
- *The model of persistence guarantees type safety.*

Orthogonal persistence is accomplished by *placing message based persistence capabilities in the root class Any*. This is a distinctive feature of this model of persistence. This model is truly object-oriented, as it is based on *message passing and inheritance*, unlike other published models that we are aware of.

We mention another project that shares similar ideas on the model of persistence for Java [21]. Although JavaSPIN does not associate the method *persist* with the class *Object*, its Java preprocessor accomplishes this effect by incorporating this method in the generated classes.

A partial specification of the predefined class *Any* is:

Class Any
Observers
persistent()
Mutators
persists()
Constraints
$\Box(\bigcirc$self.persistent() $\leftarrow$ self.persists())
End Any.

The same message based passing mechanism applies to classes, instances of the class *Class*.

4.2 Environments

The second component of the model of persistence allows name space management based on environments. As persistence is orthogonal to the type (class) hierarchy, name space management is controlled by an orthogonal mechanism. This is in sharp contradistinction to Napier [26], which has a type **env**, and Eiffel [24], which has a class ENVIRONMENT. In addition, Napier is not object-oriented, and the support for persistence in Eiffel is low-level, and not part of the language.

Name space management is controlled in Java by packages. But a package is a collection of related classes. An environment is more general in that it may contain both objects and classes. This is exactly what is needed in persistent and database applications. It is also a consequence of orthogonality. In particular, a database schema is an environment. Indeed, it contains both types and objects (typically collections) of those types.

An environment is simply a set of (*identifier*,*object*) bindings. Objects may belong to any level, i.e., they may be classes or objects. All identifiers in an environment are, of course, distinct. Predefined bindings that cannot be changed include those that apply to *Class*, *Any*, standard primitive types, collection classes, query class, etc. The root of persistence is the environment *Main* which cannot be dropped. *Main* always contains bindings for *Class* and *Any*.

An environment establishes a *scope* which can be dynamically created, extended or shrunk. When an object is promoted to longevity by a *persists* message, a new binding is introduced for this object in the currently valid scope. Scopes can be nested in such a way that the classical scoping rules apply: bindings in an inner scope are invisible in its outer scope, and bindings in the outer scope are visible (unless redefined) in the inner scope.

When an environment is open, all its bindings are made visible, so that objects in the environment can be referred to just by their names. There is no difference in referencing transient objects declared in a class and persistent objects.

Opening a scope is accomplished by a **With** block which is illustrated below. **With** block is also a unit of user interaction with the environment consisting of a collection of objects and their classes. Similar effects are accomplished in Java by package import clauses.

```
With Main
    Class ThreeDObject
    ...
    End ThreeDObject;
    Class MovingObject
    Inherits ThreeDObject
    ...
    End MovingObject;
    Environment MyEnvironment;
    With MyEnvironment
    Class Shuttle
    Inherits Moving Object
    ...
    End Shuttle;
    MyShuttle: Shuttle
    End MyEnvironment
End Main.
```

The outer **With** block opens the main environment. It then creates two classes *ThreeDObject* and *MovingObject* in the main environment, the second one derived by inheritance from the first. Another environment *MyEnvironment* is also created within the main environment. The inner **With** block introduces two bindings in the inner environment. *Shuttle* is bound to a class derived from the class *MovingObject* and *MyShuttle* is bound to an object of class *Shuttle*.

5 Persistence Implementation Architecture

Compiled *MyT* code runs on the PJama Virtual Machine. The PJama Virtual Machine is an extension of the Java Virtual Machine with persistence capabilities. PJama ([11], [20]) provides an interface *PJStore* and its implementation for managing persistent objects in Java. A simplified *PJStore* interface has the following form:

```
interface PJStore
{   ...
    public static PJStore getStore();
    public void newPRoot(String,Object);
    public void setPRoot(String,Object);
    public Object getPRoot(String);
}
```

In order to make an object persistent, an object representing the persistent store must be first made available by the method *getStore*. The method *newPRoot* is then invoked on this *PJStore* object. Its arguments are the name of the object and the object itself. The effect of invoking *newPRoot* is to introduce a binding of the argument name to the argument object in the persistent store. This promotes the argument object to persistence and makes it accessible by the name provided as the first argument of *newPRoot*. Furthermore,

all objects referred to directly or indirectly by the newly promoted object are also promoted to longevity. PJama thus supports *persistence by reachability* or *transitive persistence*.

PJama's model of persistence is *orthogonal*. However, the model of persistence in *MyT* is more sophisticated. Its distinctive features include *persistence capabilities in the root class* and *hierarchical name space management*. The name space management model allows references to transient and persistent objects without differentiation. This more sophisticated model of persistence has been implemented by developing a library of classes which is, of course, not only useful as part of *MyT*'s runtime system, but also as an addition to PJama.

The limitation of the PJama's flat name-space has been overcome by mostly ignoring its naming scheme altogether. For all practical purposes, we make PJama aware of only one named object, an instance of class *Environment* we call *Main*. *Environment* is *not* a public class. The underlying persistence implementation architecture is naturally completely hidden from *MyT* users.

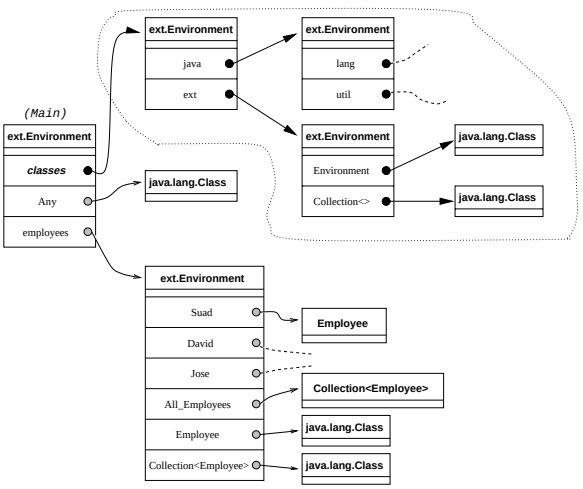

Fig. 1. *Illustration of a hypothetical environment graph of persistent objects in the MyT naming scheme*

In order to implement the class *Environment*, we use one of Java's well-known classes, *Hashtable*. PJama's reachability capabilities take care of the rest; that is, anything reachable from *Main* is, of course, persistent.

Figure 1 illustrates what a hypothetical naming graph may look like in this scheme. Note that standard Java classes appear in the *Main* environment except that their names reflect Java conventions related to packages. Other classes intended for general use, such as *Collection*, should also be in *Main*.

```
class Environment
{    Environment getParent();
     boolean isParent (Environment env);
     boolean isRoot();
     void addBinding (String name, Object obj);
     void removeBinding (String name);
     Object lookup (String name);
     void with(); // open a scope
     void close(); // close a scope
     private Hashtable aHashtable;
}
```

The class *Environment* is not directly accessible to the users. The methods of the class *Environment* are invoked indirectly through static methods of another class, called *Scoping*, which is public.

The implementation of *Environment* is actually quite straightforward. Methods *addBinding* and *removeBinding* simply invoke *Hashtable.add* and *Hashtable.remove*, respectively. Method *getParent* returns a reference to the parent environment, or *null* if invoked on the main environment. Method *lookup* is implemented recursively as follows:

```
public Object lookup (String name)
{ Object obj = aHashtable.get(name);
     if (obj == null && isRoot() == false)
          obj=getParent().lookup (name);
     return obj;
}
```

A Java programmer would normally use the *PJStore* interface to manage persistent objects. In the underlying persistent implementation architecture, objects are made persistent by invoking the method *Scoping.addBinding*. In the *MyT* user model, persistence capabilities are associated with the root class *Any*. The class *Any* has a method named *persists* which promotes its receiver object to longevity. The method *persists* is implemented simply as follows.

```
void persists (String name)
{ Scoping.addBinding (name, this);
}
```

6 Implementing F-Bounded Polymorphism

6.1 The Technique

There are several general techniques for implementing parametric polymorphism in an object oriented programming language:

- Creation of specialized code for each instantiated class. This is also known as a *heterogeneous translation* [27].

- Creation of generic code shared by all instantiated classes. The code is ready to be executed without modification. This is also known as a *homogeneous translation* [27].
- Creation of generic code with tags. The code is modified upon class loading of instantiated classes. This approach has been reported in [9] and it is referred to as *load-time expansion* or *load-time instantiation.*

C++ uses the first technique through substitution of the actual type (class) parameters in the template source code during compilation. It is arguable whether Java or MyT could ever implement any form of textual substitution, since the design philosophy of both languages does not require the source code of external classes to be available during compilation. Nevertheless, a heterogeneous translation can be achieved in these systems by having the compiler create a specialized class file from a template every time a generic class is instantiated. An obvious shortcoming of this approach is that it may unnecessarily clutter up the file system with an excessive number of files that differ only in the actual type parameters.

The designers of Pizza — a superset of Java with parametric polymorphism and other features — claim that they have achieved both homogeneous and heterogeneous translations in their system [27]. It seems, however, that the current distribution supports a homogeneous translation only. This is in fact the preferred approach since it only requires modification of the compiler (which is always necessary) and does not generate redundant class files.

A homogeneous translation is accomplished in Pizza by generating a valid Java class file from a parametric class with the bound type in place of the type parameter. This technique is possible because Java is based on single dispatch, so that the types of arguments of a message do not affect the selection of the method at run-time.

Load-time instantiation, which is the proposal found in [9], is attractive because of its simple implementation. The changes to the compiler are in all likelihood minimal, as well as the extensions of the $ClassLoader$ class. Furthermore, this approach does not clutter up the file system. Note that the Java technology allows the $ClassLoader$ class to be extended so that it can deal with class files representing parametric classes. This feature has been used in [9], in our project, and in other Java extending projects.

The main problem of load-time instantiation as used in [9] is that it requires class files extended with appropriately encoded type constraints. These class files are not valid Java class files, and are transformed into valid Java class files by the extended loader. There are many potential problems with two forms of class files, one of which is not legal.

The technique presented in this paper offers two advantages with respect to the above two. Class files generated from parametric classes are always valid Java class files, as in Pizza. In addition, the type information about instantiated generic classes is specific and available at run-time, which is not the case in Pizza.

In the source code of a MyT class specification, both persistent and transient (local) objects are treated equally. That is, objects are referred to by their names,

and there is no need to invoke a method that retrieves an object from the store. In PJama the programmer must explicitly invoke a method named *getPRoot* to retrieve an object from the store [11]. All a Java compiler can assume is that the object returned by this method is an instance of *Object*.

In *MyT* the compiler must be aware of the precise types of persistent objects in order to type-check the source code correctly. This is not a problem with instances of non-generic classes, as each object always refers to the class from which it was instantiated. But for objects of *instantiated parametric classes* the compiler must be aware not only of the generic class, but also of the instantiation parameters, or else proper type-checking cannot be carried out.

Of course, the Pizza compiler also requires this information to be available, but the difference lies in how the actual parameters are known to the compiler. In Pizza, the instantiation parameters are obtained from the source code being compiled. In fact, only the bound type is available from the class file.

In *MyT*, the source code of an instantiation is not necessarily available for persistent objects. This type-specific information is available for the instantiated parametric classes from their class objects in the persistent store. This is the crucial difference between Pizza and *MyT*. *MyT* has been designed with a persistent store in mind, and Pizza has not. This is why *MyT* solves the problem of specific type information without affecting the validity of the Java class file representation.

In our implementation, for every parametric class the *MyT* compiler creates a generic *abstract* class file. In this file, each formal type parameter is substituted by its bound type, as is done in Pizza. The class file also contains information about the actual position of each formal type parameter, which is required by the compiler to be able to correctly type-check instantiations of the generic class.

For classes obtained by instantiation of the parametric class, this technique also includes the information about the actual type parameters. This information is included in the *class name*.

The Java *ClassLoader* is extended as in [9] to read the generic class file and create appropriately instantiated class objects as explained below. This technique allows implementation of both bounded and F-bounded parametric polymorphism.

A significant difference between our approach and that of Pizza is that each instance of a generic type refers to a class whose name contains the actual type parameters of the instantiation. In Pizza, there is really no way to know — at run-time — what the actual type parameters are. Only the upper bound is available.

6.2 Class File Representation

A Java class file marked *abstract* is generated by the compiler. This class file is used as a kind of template for the classes instantiated from the parametric class. This technique was chosen for two reasons.

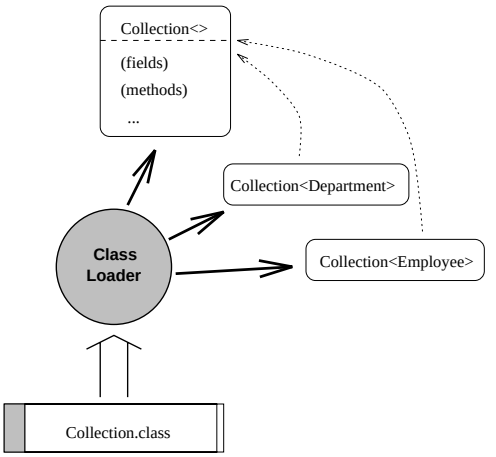

Fig. 2. *Extended class loader handles parametric classes*

- Abstract class files are available in the Java Virtual Machine. A class file representing an abstract class is generated from a generic class substituting the bound types for the type parameters. This way we obtain a perfectly valid class file.
- Abstract classes cannot be instantiated, and thus our particular representation has no effect on the execution model of the Java Virtual Machine.

None of the actual methods in this abstract class are *in fact* abstract. These methods are completely implemented by instantiation using the bound types in place of the type parameters.

The extended class loader is used to create specific class objects from the abstract template class. These class objects can take one of two possible forms, depending upon whether it is more desirable to optimize memory usage or speed of execution.

- The first form (see fig. 3) is a nearly empty subclass of the abstract class with specific type information for the actual parameters contained in the name of this extended class.
- The second form (see fig. 4) is a complete instantiation of the parametric abstract class file, with the name of the class as in the first method, and with abstract flags cleared.

The first version has the disadvantage that it adds one level of hierarchical look-up to all messages passed to an object of an instantiated parametric class, slowing execution at run time. The second implementation version has a performance advantage over the first form by eliminating the extra level of look-up in method dispatch. It has a disadvantage over the first form because it requires more space for Java class objects. This could be a particular disadvantage if the

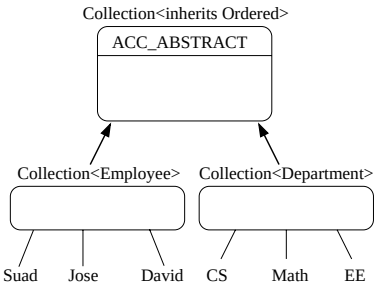

Fig. 3. *Space efficient implementation of parametric polymorphism*

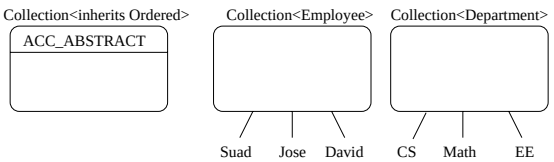

Fig. 4. *Time efficient implementation of parametric polymorphism*

same parametric class were to have numerous specific classes instantiated from it, each with different actual type parameters.

Note that class files for instantiated parametric classes are never constructed; only the class file of the parametric class exists. This is obviously an advantage of this technique. In addition, because of the technique for providing the actual type parameters for instantiated parametric classes, such files would not be valid Java class files.

The actual type information is provided in the name of the instantiated parametric class, as one might naturally expect. For example, if the name of a parametric class file is *Collection.class*, this name will appear in the class object created by the extended loader as $''Collection <>''$. An example of the name of a specific instantiation of this class is $''Collection < Employee >''$, where *Employee* is the actual type parameter. If the bound type of the type parameter of the class *Collection* is *Ordered*, the compiler will, of course, check that *Employee* extends *Ordered*. The fact that these names are not standard Java class files names for the Java compiler does not matter, as these names will be reflected only in the main memory and in the persistent store.

Consider now the actions of the extended class loader. The class loader is invoked to load a class with a given name. The standard loader must be extended in order to recognize the names of instantiated parametric classes. When such a name is detected, the abstract file representing the parametric class is loaded. This is a perfectly valid class file, as the bound type appears in place of the

formal parameter. Because of the Java single method dispatch mechanism, this fact has no effect at execution time. In addition, the extended class loader also creates a specific class object (unless already created) according to one of the two possible options explained above.

7 The Compiler

7.1 Managing Class Files

The *MyT* compiler has a collection of classes for managing Java class files. These classes allow the compiler to interpret the contents of Java class files, as well as to construct them. The latter includes byte code generation for methods.

A Java class file is carefully constructed to be in complete compliance with the Java class file structure specification, such that it passes all verification tests at load and run times. It also passes the structural criteria and static constraints tests of the independent Java class file verifier. These verifier tests are stringent and ensure before run time that there will be no operand stack overflows or underflows, that loads from and stores to all local variables are valid, and that only valid types of arguments are used for all Java Virtual Machine instructions.

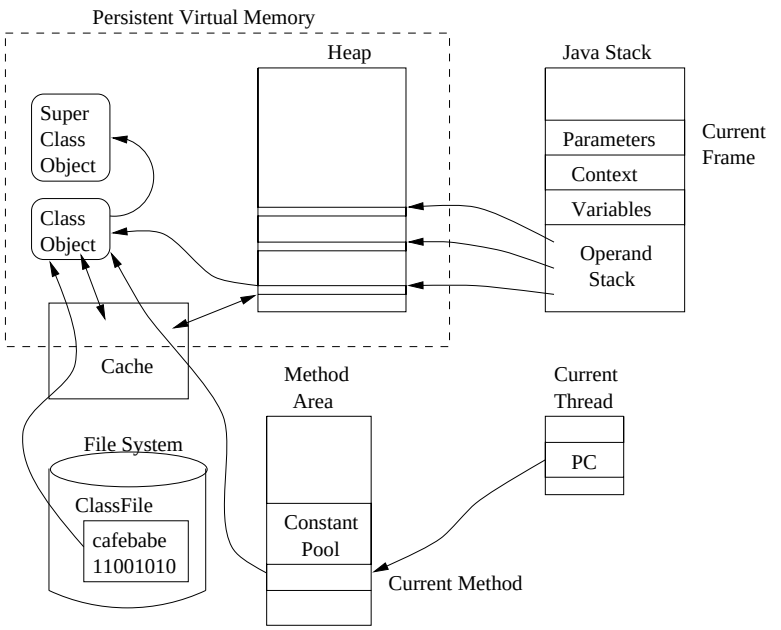

Fig. 5. *The overall architecture of the MyT system*

In PJama there exist both persistent *Class* objects and class files. The reason for this redundancy is compatibility with the Java Virtual Machine. In our system, accessing the *Class* object from the persistent store is attempted first. If the persistent *Class* object in fact exists, Java Core Reflection [19] is used to access the information on fields, methods and constructors of this class. Otherwise, the associated Java class file is located and the required information is collected from the file.

7.2 Run-Time Model

The run-time model for constraint evaluation is stack-oriented. It makes use of the Java stacks. A method invocation creates a new stack frame. A stack frame contains an operand stack, which is used in evaluating constraints as explained below. The run-time architecture of the system is presented in figure 5.

In a message $a.f(a_1, a_2, ..., a_n)$, $a, a_1, a_2, ..., a_n$ are terms which, when evaluated, produce object identifiers on the top of the operand stack. Before the selected method for f is invoked, $n+1$ object identifiers (the receiver and n arguments) are placed on the operand stack. Method lookup is based on the Java single dispatch mechanism. If f is a constructor, the result of method invocation will be the object identifier of the constructed object placed on the top of the operand stack after removal of $n+1$ top-level entries. If f is an observer, boolean (in fact Java int) is placed on the top of the operand stack.

Unary observers in most cases correspond to the fields of the underlying object state. n-ary observers are more subtle to implement. For collection classes (those derived from a predefined class *Collection*), an n-ary observer is implemented as a relation associated with the collection object. This relation is called an access relation [2] and consists of tuples of object identifiers for which the observer evaluates to true. Updates to the underlying collection must therefore be reflected in the associated access relations. A compressed representation of an access relation may be a dynamic index. Further details are elaborated in [2].

7.3 Evaluating Constraints

– *Initial constraints*
The simplest form of a constraint is the *initial* constraint, with the form $self.p(a_1, a_2, ..., a_n) \leftarrow$, where p is an observer and $a_1, a_2, ..., a_n$ are ground terms (i.e. terms with no variables). This form merely provides the values for setting the underlying state of the observer in the head of the implication.

– *Class invariants*
Class invariants test whether the state of the object is acceptable. They are of the form:

$\Box(self.p(a_1, a_2, ..., a_n) \leftarrow$
$self.p_1(a_{11}, a_{12}, ..., a_{1n}),\ self.p_2(a_{21}, a_{22}, ..., a_{2n}),\ ...,\ self.p_k(a_{k1}, a_{k2}, ..., a_{kn}))$

where p, p_1, p_2, ..., p_k are observers.

– *Preconditions*
Precondition constraints check that the state of the object is acceptable for the execution of a mutator, given a list of arguments to that mutator. They are of the form

$$\Box(self.p(a_1, a_2, ..., a_n) \leftarrow self.m(a_{m1}, a_{m2}, ..., a_{mn}))$$

where p is an observer and m is a mutator.
– *Postconditions*
Postcondition constraints test whether the final state of the object is acceptable following the execution of a mutator, given the set of arguments to that mutator. They are of the form

$$\Box(\bigcirc self.p(a_1, a_2, ..., a_n) \leftarrow self.m(a_{m1}, a_{m2}, ..., a_{mn}))$$

where p is an observer and m is a mutator.
The final state of the object following execution of a mutator must also meet the criteria of the *class invariants*, which are tested before the constraints are tested.

– *Transition constraints*
Transition constraints actually specify changes of the state of an object, provided that the changes pass the criteria of the *precondition*, *postcondition* and *class invariant* constraints, as well as the criteria specified by the *transition* constraint itself. The form of a *transition* constraint is:

$$\Box(\bigcirc\ self.p(a_1,\ a_2,\ ...,\ a_n) \leftarrow self.m(a_{m1}, a_{m2}, ..., a_{mn}),\ self.p_1(a_{11}, a_{12}, ..., a_{1n}),\ self.p_2(\ a_{21},\ a_{22},\ ...,\ a_{2n}),\ ...,\ self.p_k(a_{k1},\ a_{k2},\ ...,\ a_{kn}))$$

where p, p_1, p_2, ..., p_k are observers, and m is a mutator.

Due to space limitations, we describe briefly just the evaluation of a transition constraint. When evaluating a transition constraint, the observers in the body of a transition clause are invoked in turn. If any observer returns *false*, then the transition fails. If all observers return *true*, then a temporary clone of the object is created, and the state of the observer in the head of the implication is set appropriately. The *class invariant* constraint and *postcondition* constraint methods are invoked on the clone object. If any of these constraints return *false*, then the transition fails. If all return *true*, then the transition is committed, and the state of the observer in the head of the implication in the original object is set accordingly. Of course, the overall result of a constraint evaluation is also pushing the Java integer representation of the computed logical value on top of the operand stack.

8 Conclusions

Our goal was to investigate whether the existing Java technology can be suitably extended by very non-traditional components targeted to applications in which the Java language itself has serious limitations. Our targeted application areas are database and persistent object systems. Java itself has problems in

dealing with such applications since it does not support persistence, parametric polymorphism and assertions (constraints). Our goal was to accomplish this extension without changing the Java language and its Virtual Machine.

The component extending the Java technology designed and implemented in this project is in fact a statically typed, declarative object-oriented language MyT. The language features a high-level model of persistence, with orthogonality and reachability. Its type system is much more sophisticated than the Java type system. In particular, it supports F-bounded polymorphism. The language is in fact a constraint language, sufficiently expressive to allow assertions typical for object-oriented languages (preconditions, postconditions, class invariants, history properties). The language allows references to Java interfaces and Java classes. It supports name space management which extends similar facilities both in Java and in its persistent extensions such as PJama.

This result shows that the Java Virtual Machine is in fact a suitable implementation platform even for very non-traditional object-oriented languages. However, an extension of the Java Virtual Machine is still required in order to support persistence.

Specific techniques presented in the paper for implementing a high-level model of persistence and F-bounded polymorphism are in our opinion of general interest. The advantages of the model of persistence is that it is truly object-oriented because it is based on message passing and inheritance. It is also high-level, and naturally supports orthogonality and transitivity of persistence. In particular, the name space management model gives the impression that there is no difference between users' references to transient and persistent objects.

The advantages of the implementation technique for bounded and F-bounded polymorphism is that it makes use of valid Java class files and provides specific type information for classes obtained by instantiating parametric classes. This specific type information, missing in some other techniques, is important for systems that make use of Java Core Reflection [19]. It is also important for models of persistence that are equipped with a high-level model of name space management, like the one proposed in this paper. Last but not least, more sophisticated (multiple) method dispatch techniques will benefit from this technique.

We expect that the idea to generate Java byte code from constraints will be of interest to implementors of declarative (query in particular) object-oriented languages on top of the Java Virtual Machine. Java OQL [15] would be a particularly important example.

Parametric polymorphism is not the only desirable feature missing from the Java type system. A more expressive type system based on self types avoids problems in expressing binary methods in a natural manner ([1], [13]). The type system of the full-fledged MyT is in fact more general than the type system presented in this paper because it supports self types. Type safety is accomplished using a technique called constrained matching [4]. Unlike the techniques of type systems, constrained matching delegates problems that cannot be handled by the type system to the temporal constraint system. In addition, its run-time model is based on a limited form of multiple dispatch.

Although Java is based on single dispatch, the Java Virtual Machine makes it possible to implement the form of multiple dispatch technique required to support self types. This is due to the wealth of information (type information in particular) contained in a Java class file. But integrating this extension with the Java environment causes nontrivial problems in order to avoid corrupting the Java type system. This issue is left as an open problem for future research.

Due to space limitations, the implementation architecture for persistent collections has not been elaborated in this paper. A persistent collection may be equipped with multiple access paths (access relations [2]) that are implemented using dynamic indices or extensible hashing. Orthogonal persistence and persistence by reachability are thus a must in order to be able to manage correctly this complex structure of collection objects.

References

1. Abadi, M., Cardelli, L.: On Subtyping and Matching, Proceedings of ECOOP '96, *Lecture Notes in Computer Science* **1098**. Springer-Verlag (1996) 145-167. 217, 219, 231
2. Alagić, S.: A Temporal Constraint System for Object-Oriented Databases, Constraint Databases and Applications, Proceedings of CDB '97 and CP '96 Workshops, *Lecture Notes in Computer Science* **1191**. Springer-Verlag (1997) 208-218. 213, 215, 229, 229, 232
3. Alagić, S.: The ODMG Object Model: Does it Make Sense? Proceedings of the OOPSLA '97 Conference. ACM (1997) 253-270. 213, 214, 214, 214, 216, 218, 218
4. Alagić, S.: Constrained Matching is Type Safe, Proceedings of the 6th Database Programming Language Workshop (DBPL), 1997, *Lecture Notes in Computer Science*. Springer-Verlag (1998) (to appear). 213, 214, 215, 217, 231
5. Alagić, S., Alagić, M.: Order-Sorted Model Theory for Temporal Executable Specifications, *Theoretical Computer Science* **179** (1997) 273-299. 215
6. Alagić, S.: A Statically Typed, Temporal Object-Oriented Database Technology, *Transactions on Information and Systems* **78**. IEICE (1995) 1469-1476. 213
7. Alagić, S., Sunderraman, R., Bagai, R.: Declarative Object-Oriented Programming: Inheritance, Subtyping and Prototyping, Proceedings of ECOOP '94, *Lecture Notes in Computer Science* **821**. Springer-Verlag (1994) 236-259.
8. Alagić, S.: F-bounded Polymorphism for Database Programming Languages, Proceedings of the 2nd East-West Database Workshop, *Workshops in Computing*. Springer-Verlag (1994) 125-137. 214, 218
9. Agesen, O., Freund, S., Mitchell, J. C.: Adding Type Parameterization to Java, Proceedings of the OOPSLA '97 Conference. ACM (1997) 49-65. 213, 224, 224, 224, 224, 225
10. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Zdonik, S.: The Object-Oriented Database System Manifesto, Proceedings of the First Object-Oriented and Deductive Database Conference, Kyoto (1989) 223-240. 212
11. Atkinson, M., Daynes, L., Jordan, M. J., Printezis, T., Spence, S.: An Orthogonally Persistent Java^{TM}, ACM SIGMOD Record **25** (4) (1996) 68-75. 213, 213, 213, 221, 225
12. Atkinson, M., Morrison, R.: Orthogonally Persistent Object Systems, *VLDB Journal* **4** (1995) 319-401. 216

13. Bruce, K., Schuett, A., van Gent, R.: PolyTOIL: a Type-Safe Polymorphic Object Oriented Language, Proceedings of ECOOP '95, *Lecture Notes in Computer Science* **952**. Springer-Verlag (1996) 27-51. 217, 231
14. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J. C.: F-bounded Polymorphism for Object-Oriented Programming, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture. ACM (1989) 273-280. 214, 218
15. Cattell, R. G. G., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., Wade, D.: *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann (1997). 213, 213, 214, 216, 231
16. Cooper, R., Kirby, G.: Type-Safe Linguistic Run-time Reflection: A Practical Perspective, Proceedings of the 6th Int. Workshop on Persistent Object Systems, *Workshops in Computing*. Springer-Verlag (1994) 331-354. 216
17. Gosling, J., Joy, B., Steele, G.: *The Java^{TM} Language Specification*. Addison-Wesley (1996). 212
18. Gawecki, A., Matthes, F.: Integrating Subtyping, Matching and Type Quantification: A Practical Perspective, Proceedings of ECOOP '96, *Lecture Notes in Computer Science* **1098**. Springer-Verlag (1996) 25-47. 219
19. Java Core Reflection, JDK 1.1, Sun Microsystems (1997). 229, 231
20. Jordan, M.: Early Experiences with Persistent Java^{TM}, Proceedings of the First Int. Workshop on Persistence and Java, SUN Microsystems Laboratories (1996). 221
21. Kaplan, A., Myrestrand, G. A., Ridgeway, J. V. E., Wileden, J. C.: Our SPIN on Persistent Java^{TM}, Proceedings of the First Int. Workshop on Persistence and Java, SUN Microsystems Laboratories (1996). 219
22. Liskov, B., Wing, J. M.: A Behavioral Notion of Subtyping, *ACM Transactions on Programming Languages and Systems* **16** (1994) 1811-1841. 212
23. Lindholm, T., Yellin, F.: *The Java^{TM} Virtual Machine Specification*. Addison-Wesley (1996). 213
24. Meyer, B.: *Eiffel: the Language*. Prentice-Hall (1992). 212, 218, 220
25. Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall (1997). 212, 218
26. Morrison, R., Brown, A. L., Connor, R., Dearle, A.: Napier88 Reference Manual, Universities of Glasgow and St. Andrews Technical Report PPRR-77-89 (1989). 220
27. Odersky, M., Wadler, P.: Pizza into Java: Translating Theory into Practice, Proceedings of the POPL Conference. ACM (1997) 146-159. 213, 223, 224, 224
28. Pierce, B. C.: Bounded Quantification is Undecidable, Proceedings of the POPL Conference. ACM (1993) 305-315. 219